Dissertation

Top-k Semantic Caching

by
Christoph Ehlers

submitted to
Department of Informatics and Mathematics,
University of Passau

April, 2015

Supervisor:
Professor Dr. Burkhard Freitag
1. Reviewer: Professor Dr. Burkhard Freitag
   Chair of Information Management
   University of Passau
   Innstraße 43
   94032 Passau, Germany
   E-mail: Burkhard.Freitag@uni-passau.de
   Web: http://www.im.uni-passau.de/

2. Reviewer: Professor Dr. Werner Kießling
   Chair of Databases and Information Systems
   Institute of Computer Science
   University of Augsburg
   Universitätsstraße 6a
   86159 Augsburg, Germany
   E-mail: kießling@informatik.uni-augsburg.de
   Web: http://www.informatik.uni-augsburg.de/de/lehrstuehle/dbis/
Abstract

The subject of this thesis is the intelligent caching of top-k queries in an environment with high latency and low throughput. In such an environment, caching can be used to reduce network traffic and improve response time. Slow database connections of mobile devices and to databases, which have been offshored, are practical use cases.

A semantic cache is a query-based cache that caches query results and maintains their semantic description. It reuses partial matches of previous query results. Each query that is processed by the semantic cache is split into two disjoint parts: one that can be completely answered with tuples of the cache (probe query), and another that requires tuples to be transferred from the server (remainder query).

Existing semantic caches do not support top-k queries, i.e., ordered and limited queries. In this thesis, we present an innovative semantic cache that naturally supports top-k queries. The support of top-k queries in a semantic cache has considerable effects on cache elements, operations on cache elements — like creation, difference, intersection, and union — and query answering. Hence, we introduce new techniques for cache management and query processing. They enable the semantic cache to become a true top-k semantic cache.

In addition, we have developed a new algorithm that can estimate the lower bounds of query results of sorted queries using multidimensional histograms. Using this algorithm, our top-k semantic cache is able to pipeline partial query results of top-k queries. Thereby, query execution performance can be significantly increased.

We have implemented a prototype of a top-k semantic cache called IQCache (Intelligent Query Cache). An extensive and thorough evaluation with various benchmarks using our prototype demonstrates the applicability and performance of top-k semantic caching in practice. The experiments prove that the top-k semantic cache invariably outperforms simple hash-based caching strategies and scales very well.
Contents

Abstract i

Contents iii

I. Fundamentals 1

1. Introduction and Motivation 3
   1.1. Semantic Caching 3
   1.2. Running Example 5
      1.2.1. Three Queries 6
      1.2.2. Database Instance 7
      1.2.3. A Motivation for Top-k Semantic Caching 7
   1.3. Problem Description 13
   1.4. Approach 13
   1.5. Assumptions 14
   1.6. Contributions 15
   1.7. Overview 16

2. Related Work of Semantic Caching 19
   2.1. Semantic Caching of SQL Queries 19
   2.2. Semantic Caching of Non-SQL Queries 22
      2.2.1. XML Databases 22
      2.2.2. Web Queries and Information Retrieval 23
      2.2.3. Spatial Data 23
   2.3. Other Related Caching Techniques 23
      2.3.1. Logical Caching 24
      2.3.2. Table Caching 24
      2.3.3. Answering Queries Using Views 25
      2.3.4. Caching of Top-k Queries with Linear Score Functions 25

3. Satisfiability 27
   3.1. Satisfiability and Domains 27
      3.1.1. Variables and Domains 28
3.1.2. The Varchar Domain ........................................... 28
3.1.3. From SAT to SMT ............................................ 30
3.2. SMT Solver ...................................................... 31
  3.2.1. Solver Selection .......................................... 31
  3.2.2. Hybrid Solver ............................................. 33
3.3. Three-valued Logic and SQL ................................. 41
  3.3.1. Conjunction, Disjunction, Negation ..................... 41
  3.3.2. Isunknown ............................................... 41
  3.3.3. Partial Equivalence .................................... 44
3.4. Satisfiability and Three-valued Logic ...................... 46
  3.4.1. Overlaps ................................................. 46
  3.4.2. Subsumptions ............................................ 47

II. Top-k Semantic Cache Design .................................. 51

4. Segments .......................................................... 53
  4.1. Segment Definition ......................................... 53
    4.1.1. Definition ............................................... 53
    4.1.2. Conditions .............................................. 56
    4.1.3. Order By Condition of Lexicographical Orderings ... 60
    4.1.4. Orderings Using Arbitrary Score Functions ........... 63
  4.2. Set Operations on Segments ................................ 67
    4.2.1. Creation of a New Segment ............................ 67
    4.2.2. Difference of Two Segments ............................ 78
    4.2.3. Intersection of Two Segments ......................... 92
    4.2.4. Union of Two Segments ................................ 96
    4.2.4.1. The General Case .................................... 96
    4.2.4.2. A Special Case ...................................... 101

5. Pipelining .......................................................... 107
  5.1. Multidimensional Histograms .............................. 107
    5.1.1. Definition .............................................. 108
  5.2. Histograms for the Estimation of Query Bounds ........... 109
    5.2.1. Properties of Histograms for the Estimation of Query Bounds ............................................. 109
    5.2.2. NP-Completeness of Optimal Histogram Selection . 112
  5.3. Estimating Query Bounds with Histograms ................ 117
    5.3.1. The Idea ............................................... 117
    5.3.2. A Sample Multidimensional Histogram ................ 120
    5.3.3. The Algorithm ......................................... 123
    5.3.4. Two Examples .......................................... 130
Contents

8.5.2. Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
  8.5.2.1. Cache Size and Hit Rate . . . . . . . . . . . . . . . . . . 190
  8.5.2.2. Cache Size and Execution Time . . . . . . . . . . . . . 191
  8.5.2.3. Server Latency and Execution Time . . . . . . . . . . . 191
  8.5.2.4. Throughput and Execution Time . . . . . . . . . . . . . 193
  8.5.2.5. Satisfiability Checks and Execution Time . . . . . . . 193
  8.5.2.6. Estimation Performance and Execution Time . . . . . . 195
  8.5.2.7. Scale and Hit Rate . . . . . . . . . . . . . . . . . . . . . . 197
  8.5.2.8. Scale and Execution Time . . . . . . . . . . . . . . . . . 199
  8.5.2.9. Scale and Number of Considered Segments . . . . . . . 199
  8.5.2.10. Scale and Number of Used Segments . . . . . . . . . . . 201

8.6. MonArch 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 203
  8.6.1. Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 203
  8.6.2. Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 204
    8.6.2.1. Cache Size and Hit Rate . . . . . . . . . . . . . . . . . . 204
    8.6.2.2. Cache Size and Execution Time . . . . . . . . . . . . . 205

8.7. Discussion and Comparison of Results . . . . . . . . . . . . . . . . . . . . 207

9. Conclusion 209
  9.1. Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209
    9.1.1. Prefetching . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 209
    9.1.2. Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 210
    9.1.3. The Mobile Use Case . . . . . . . . . . . . . . . . . . . . . . . . . 211
  9.2. Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 214

A. Histogram Creation with Window Functions 217

Bibliography 221

List of Figures 243

List of Tables 247

List of Algorithms 249

List of Listings 251

List of Examples 253

List of Definitions 255

List of Propositions, Theorems and Corollaries 257
Part I.

Fundamentals
1. Introduction and Motivation

In an environment with high latency and low throughput, caching can be used to reduce network traffic and improve response time [CFLS91, DR92]. This is particularly true for a mobile computing environment. A cache can be used to intelligently store data of mobile applications (e.g., map apps). It can also be used to cope with a slow connection and connection disruptions when checking and entering data on location. Examples are taking inventory in stores, maintenance and machine care (e.g., cars, ships) and work progress in production.

Data caching also becomes important if the database is distant. This may be due to offshoring (e.g., a transatlantic connection) or the fact that the database has been moved into the cloud. Caching is also necessary if the system is a globally distributed client-server architecture with a central database to unburden the central server (e.g., a logistics company that operates world-wide). Data caching is viewed as a key technique for efficient application servers [Moh02]. Furthermore, it is possible to save energy through caching [LRS+12].

Semantic caching has proven to be a very successful caching scheme for mobile computing environments and distributed systems. It provides better performance than page caching [CFZ94, FCL93] or tuple caching [DFJ+96]. In addition, semantic caching achieves a significant workload reduction in distributed systems [RDK03]. When semantic caching was first proposed, solvers for logical expressions over various domains, i.e., SMT solvers, did not exist. Only in the last ten years, since the introduction of the annual SMT-COMP competition [SMT05, CSW14], efficient SMT solvers have been developed and continuously been improved. Now is the time to revisit semantic caching and apply SMT solvers to it. We are the first to do so.

1.1. Semantic Caching

We define semantic caching by the following main properties [DFJ+96, KB96, JAT+06]:

Query cache: A semantic cache is a query-based cache that caches query results. Typically, the elements of a semantic cache are called segments or regions.

Description with logical expressions: The semantic cache maintains logical expressions that describe the tuples contained in its segments or rather the area that is occupied by
1. Introduction and Motivation

the segment. This approach makes a semantic cache a logical cache and distinguishes it from a page caches or tuple caches.

Disjointness: In a semantic cache, no tuple is stored more than once. Hence, redundancy in the cache is reduced. A semantic cache can achieve this property by ensuring that all segments are disjoint. This approach yields another advantage in query processing: If a segment is found that overlaps with a given query, all tuples of the overlap certainly contribute to the result of the query, because they cannot be contained in any other segment.

Partial answering: A semantic cache reuses partial matches of previous query results. Each query that is processed by the semantic cache is split into two disjoint parts: one that can be completely answered with tuples of the cache (probe query), and another that requires tuples to be transferred from the server (remainder query). Figure 1.1 illustrates this process.

Figure 1.1.: Semantic caching with probe query and remainder query
1.2. Running Example

This thesis uses a running example, which exemplifies and illustrates the algorithms and propositions that are introduced. The running example is a simplified blogging platform (see figure 1.2). On this platform, authors can compose posts, which can be assigned to various post categories. Each post may contain a number of images, which, on their part, can be assigned to image categories.

For each author, we store a unique id, their name, their total number of posts and a score. The latter is calculated by a formula that uses the likes and dislikes of the author’s posts. A post has a unique id, a title, its content and a number of likes and dislikes. In addition, the time and date of creation and last change are stored. Posts may contain images. They are referenced by their URI. A post must not use the same image twice. When an image is included in a post, its preferred width and height should be specified. With this information, the back-end system is able to scale the image before delivery to reduce network load. For different posts, naturally, different preferred width and height can be given. Both post categories and image categories are identified by their respective names.

Figure 1.2: Entity-relationship model of the running example using Chen's notation [Che76]
1. Introduction and Motivation

Figure 1.2 contains an entity-relationship model of the running example using Chen’s notation [Che76]. Figure 1.3 shows the table definitions of the running example, which include primary key constraints, foreign key constraints and chosen data types.

author : {
    id : INTEGER,
    name : VARCHAR(40),
    score : DOUBLE,
    total : INTEGER
}

image : {
    post : INTEGER (→ post.id),
    uri : VARCHAR(50),
    width : INTEGER,
    height : INTEGER
}

group : {
    author : INTEGER (→ author.id)
}

post : {
    id : INTEGER,
    title : VARCHAR(100),
    content : VARCHAR(5000),
    likes : INTEGER,
    dislikes : INTEGER,
    created : TIMESTAMP,
    changed : TIMESTAMP,
    author : INTEGER (→ author.id)
}

imagecategory : {
    name : VARCHAR(30),
    post : INTEGER (→ image.post),
    image : INTEGER (→ image.uri)
}

Figure 1.3.: Table definitions of the running example

1.2.1. Three Queries

To exemplify most of the algorithms in this thesis, we will use a sequence of three queries called $q_1$, $q_2$ and $q_3$. All three queries are top-10 queries. They all use the relation post. They all have the same select clause, which consists of the id of the post, its number of likes, its creation date, its title and its author’s id. But the three queries do have different where conditions and use varying orderings in the order by clause.

Query $q_1$ returns the top ten most liked posts of last week (see listing 1.1). But it only considers posts with at least one hundred likes.

```sql
SELECT p.id, p.likes, p.created, p.title, p.author
FROM post p
WHERE (p.created >= LAST_WEEK) AND (p.likes >= 100)
ORDER BY p.likes DESC
LIMIT 10
```

Listing 1.1: Query $q_1$: a top-k query returning last week’s most liked posts
1.2. Running Example

Query $q_2$ fetches today’s posts (see listing 1.2). But this query only returns posts that either have at least twenty likes or have been created within the hour. The posts are ascendingly ordered by their respective id.

```
1 SELECT p.id, p.likes, p.created, p.title, p.author
2 FROM post p
3 WHERE (p.created >= TODAY)
4   AND ((p.created >= LAST_HOUR) OR (p.likes >= 20))
5 ORDER BY p.id ASC
6 LIMIT 10
```

Listing 1.2: Query $q_2$: a top-k query returning today’s posts by id

Query $q_3$ returns the top ten most liked posts that have been created within the last hour (see listing 1.3).

```
1 SELECT p.id, p.likes, p.created, p.title, p.author
2 FROM post p
3 WHERE p.created >= LAST_HOUR
4 ORDER BY p.likes DESC
5 LIMIT 10
```

Listing 1.3: Query $q_3$: a top-k query returning the most liked posts of the last hour

1.2.2. Database Instance

To execute the queries $q_1$, $q_2$ and $q_3$, we need a sample instance of the relation post of the running example (see table 1.1). The tuples have the artificially numbered ids 1 to 28. They are addressed by $μ_1, \ldots, μ_{28}$, i.e., the first tuple of relation post is called tuple $μ_1$. Note that only the columns $id$, $likes$ and $created$ are given. These are the columns that are used in the where conditions and order by clauses of the queries $q_1$, $q_2$ and $q_3$. Only these columns, we need to know to determine the result of the three queries. Table 1.1 also shows the results of queries $q_1$, $q_2$ and $q_3$ using this sample instance of the relation post.

1.2.3. A Motivation for Top-k Semantic Caching

We start with the execution of query $q_1$. It returns, in that order, the tuples $μ_3, μ_4, μ_{10}, μ_7, μ_1, μ_{19}, μ_{14}, μ_8, μ_9$ and $μ_2$ (see table 1.1). Afterward, we execute query $q_2$, which returns, in that order, the tuples $μ_7, μ_8, μ_9, μ_{10}, μ_{11}, μ_{12}, μ_{13}, μ_{14}, μ_{15}$ and $μ_{16}$ (see table 1.1). Obviously, the results of queries $q_1$ and $q_2$ overlap. The tuples $μ_7, μ_8, μ_9$ and $μ_{10}$ are contained in the results of both queries (see figure 1.4). Note that tuple $μ_{16}$ satisfies the where condition of query $q_1$, but is not loaded by the query, because it is only at position twelve of the ordering of query $q_1$. Therefore, query $q_2$ must load this tuple by itself.
1. **Introduction and Motivation**

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>q₁</th>
<th>q₂</th>
<th>q₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>µ₁</td>
<td>1</td>
<td>210</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₂</td>
<td>2</td>
<td>160</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₃</td>
<td>3</td>
<td>250</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₄</td>
<td>4</td>
<td>240</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₅</td>
<td>5</td>
<td>110</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₆</td>
<td>6</td>
<td>150</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₇</td>
<td>7</td>
<td>220</td>
<td>today</td>
<td></td>
<td></td>
<td>⁴</td>
</tr>
<tr>
<td>µ₈</td>
<td>8</td>
<td>180</td>
<td>today</td>
<td></td>
<td></td>
<td>²</td>
</tr>
<tr>
<td>µ₉</td>
<td>9</td>
<td>170</td>
<td>last hour</td>
<td></td>
<td></td>
<td>³</td>
</tr>
<tr>
<td>µ₁₀</td>
<td>10</td>
<td>230</td>
<td>last hour</td>
<td></td>
<td></td>
<td>³</td>
</tr>
<tr>
<td>µ₁₁</td>
<td>11</td>
<td>20</td>
<td>today</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₁₂</td>
<td>12</td>
<td>40</td>
<td>last hour</td>
<td></td>
<td></td>
<td>⁶</td>
</tr>
<tr>
<td>µ₁₃</td>
<td>13</td>
<td>80</td>
<td>last hour</td>
<td></td>
<td></td>
<td>⁷</td>
</tr>
<tr>
<td>µ₁₄</td>
<td>14</td>
<td>190</td>
<td>today</td>
<td></td>
<td></td>
<td>⁸</td>
</tr>
<tr>
<td>µ₁₅</td>
<td>15</td>
<td>50</td>
<td>today</td>
<td></td>
<td></td>
<td>⁹</td>
</tr>
<tr>
<td>µ₁₆</td>
<td>16</td>
<td>140</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹²</td>
</tr>
<tr>
<td>µ₁₇</td>
<td>17</td>
<td>120</td>
<td>today</td>
<td></td>
<td></td>
<td>¹⁴</td>
</tr>
<tr>
<td>µ₁₈</td>
<td>18</td>
<td>70</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₁₉</td>
<td>19</td>
<td>200</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹²</td>
</tr>
<tr>
<td>µ₂₀</td>
<td>20</td>
<td>80</td>
<td>today</td>
<td></td>
<td></td>
<td>¹³</td>
</tr>
<tr>
<td>µ₂₁</td>
<td>21</td>
<td>90</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁴</td>
</tr>
<tr>
<td>µ₂₂</td>
<td>22</td>
<td>130</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁵</td>
</tr>
<tr>
<td>µ₂₃</td>
<td>23</td>
<td>70</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁶</td>
</tr>
<tr>
<td>µ₂₄</td>
<td>24</td>
<td>30</td>
<td>last week</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>µ₂₅</td>
<td>25</td>
<td>60</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁷</td>
</tr>
<tr>
<td>µ₂₆</td>
<td>26</td>
<td>10</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁸</td>
</tr>
<tr>
<td>µ₂₇</td>
<td>27</td>
<td>50</td>
<td>last hour</td>
<td></td>
<td></td>
<td>¹⁹</td>
</tr>
<tr>
<td>µ₂₈</td>
<td>28</td>
<td>10</td>
<td>today</td>
<td></td>
<td></td>
<td>²⁰</td>
</tr>
</tbody>
</table>

Table 1.1.: Sample instance of the relation *post* of the running example
Tuple $\mu_{19}$, on the other hand, is loaded by query $q_1$, but is not used by query $q_2$, because it is only at position twelve of the ordering used by query $q_2$.

In contrast, tuple $\mu_{17}$ is needed by neither query $q_1$ nor query $q_2$, even though it satisfies both their where conditions, because it is only at position 14 for query $q_1$ or position 11 for query $q_2$, respectively.

A traditional semantic cache [KB96, DFJ+96, JAT+06] would not be of much help to exploit the overlap of query $q_1$ and query $q_2$. First, it would not be able to store the result of query $q_1$ because of the limitation. Secondly, because the query $q_2$ is sorted, the cache would need to wait for the server to transfer the remaining tuples $\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}$ and $\mu_{16}$. Thereby, the benefit of the semantic cache would be almost nullified.

However, a top-k semantic cache that naturally supports sorted and limited queries is able to store the results of queries $q_1$ and $q_2$. In addition, it is able to estimate a lower bound for the first tuple from the server in the result of query $q_2$ using a multidimensional histogram. This allows for pipelining.

The first four tuples of the result of query $q_2$, i.e., $\mu_7, \mu_8, \mu_9$ and $\mu_{10}$, have already been loaded by query $q_1$. If the cache is able to evaluate, e.g., with a multidimensional histogram, that all tuples that will have to be loaded from the server are greater than these four tuples, it can immediately return these four tuples. In this way, it is able to pipeline the result of query $q_2$ (see figure 1.5).

After query $q_1$ and then query $q_2$ have been processed, we execute query $q_3$. It returns, in that order, the tuples $\mu_{10}, \mu_{19}, \mu_9, \mu_{16}, \mu_{22}, \mu_{21}, \mu_{13}, \mu_{23}, \mu_{25}$ and $\mu_{27}$ (see table 1.1). The result of query $q_3$ overlaps with both the results of previously executed queries $q_1$ and $q_2$.
and \( q_2 \).

The overlap of the results of queries \( q_1 \) and \( q_3 \) consists of the tuples \( \mu_{10}, \mu_{19}, \mu_9 \) (see figure [1.6]).

Note that the tuples \( \mu_{16} \) and \( \mu_{22} \) also satisfy the \textit{where} condition of query \( q_1 \), but they have not been loaded by query \( q_1 \), because they are only at position 12 and 13 of the ordering of the top-10 query \( q_1 \).

\[ q_1 \]

\[ q_2 \]

\[ q_3 \]

The overlap of query \( q_2 \) and query \( q_3 \) contains the tuples \( \mu_{10}, \mu_9, \mu_{16} \) and \( \mu_{13} \) (see figure [1.7]).

Note that the tuples \( \mu_{19}, \mu_{22}, \mu_{21}, \mu_{23}, \mu_{25} \) and \( \mu_{27} \) also satisfy the \textit{where} condition of query \( q_2 \), but they have not been loaded by query \( q_2 \), because they are only at position 12 or worse of the ordering of the top-10 query \( q_2 \).
1.2. Running Example

Tuple $\mu_{12}$, however, has been loaded by query $q_2$, but it is not needed by query $q_3$, because it is at position 11 of the ordering used by query $q_3$.

In contrast, tuple $\mu_{26}$ is loaded by neither query $q_1$ nor query $q_2$, despite it satisfying both their where conditions, because it is only at position 18 for query $q_2$ or position 12 for query $q_3$, respectively.

Together, in theory, the results of query $q_1$ and query $q_2$ can contribute the tuples $\mu_{10}$, $\mu_{19}$, $\mu_{9}$, $\mu_{16}$ and $\mu_{13}$ to the result of query $q_3$. Again, this will only be possible if a cache is used that supports sorted and limited queries, e.g., a top-k semantic cache.

Once again, the first four tuples of query $q_3$, i.e., $\mu_{10}$, $\mu_{19}$, $\mu_{9}$ and $\mu_{16}$, have already been loaded by queries $q_1$ and $q_2$. If a top-k semantic cache is able to evaluate that all tuples that will have to be loaded from the server are greater than these four tuples, e.g., with a multidimensional histogram, it can return these four tuples at once. By this means, the top-k semantic cache is able to pipeline the result of query $q_3$ (see figure 1.8).

A thorough overview of the overlaps of the results of queries $q_1$, $q_2$ and $q_3$ can be seen in figure 1.9.

In conclusion, the queries $q_1$, $q_2$ and $q_3$ show the caching and pipelining potential of top-k queries that can only be exploited by a top-k semantic cache.
1. Introduction and Motivation

Figure 1.8.: Pipelining possibility of query \( q_3 \)

Figure 1.9.: Overlaps of query \( q_1 \), query \( q_2 \) and query \( q_3 \)
1.3. Problem Description

To the best of our knowledge, there exists no semantic cache that can store limited queries. In addition, there exists no semantic cache that can pipeline partial results of queries with orderings. We will attempt to design a top-k semantic cache that can do both.

Approaches that can process limited queries with orderings do exist [APT03, LNX08]. But they ignore the limit for their cache management. They send the query without the limit to the back-end database and cache the result of that unlimited query. While this approach might be utilized for server-side caching, it is not reasonable for client-side caching, because, using this approach, plenty of irrelevant, unneeded tuples are transferred through the slow connection between server and client.

The first question that we have to answer is how to model segments of a top-k semantic cache. Then, we have to introduce set operations like difference, intersection, and union that operate on these segments to facilitate cache management. In addition, we have to design an algorithm for query processing in a top-k semantic cache.

To allow for pipelining of partial results of sorted queries, we must be able to estimate a lower bound for the remaining tuples that have to be transferred from the server to the client. Hence, we need to develop a method to estimate a lower bound for the result of a given remainder query.

1.4. Approach

First, we will identify all properties that are needed to describe a segment of a top-k semantic cache. We will regard the challenge of using logical expression to specify the utilized orderings and limits. In addition, we will study and prove how to use logical expressions to describe the set of loaded or not yet loaded tuples, respectively. Based on the segment descriptions and the proven logical expressions, we will develop algorithms for set operations like difference, intersection and union on these segments. Thereby, we will consider the possibility of in-place implementations of these algorithms. Then, we will introduce cache organization, coalescing strategies, and query processing for a top-k semantic cache.

We will consider multidimensional histograms to estimate lower bounds of query results of sorted queries. We will design an algorithm that can estimate lower bounds of query results of sorted queries using aforementioned multidimensional histograms. Furthermore, we will analyze the difficulty of histogram selection under given space constraints.

We will implement and evaluate the proposed top-k semantic cache. For our experiments, we will use well-established, proven benchmarks like the Yahoo! Cloud Serving
1. Introduction and Motivation

Benchmark \([\text{CST}^{+}10]\) and the Twitter benchmark \([\text{CHBG}10]\). In addition, we will use real world practical queries of the MonArch 3 system \([\text{SWF}11]\).

1.5. Assumptions

We assume that we use a Java application that accesses a database using a JDBC driver (see figure 1.10). We assume that the connection has a high latency and a low throughput.

![Figure 1.10.: Set-up with high latency and low throughput](image)

The Java application must implement the JDBC interface to integrate the JDBC driver. Hence, we can assume that the JDBC driver is interchangeable. We assume that the top-k semantic cache is implemented as a JDBC driver. It uses the original JDBC connection to access the database. To store the data of the cache elements, it uses a second JDBC connection to an in-memory database like HSQLDB \([\text{Thed}]\) or to a fast local database like MySQL \([\text{Ora}]\). Therefore, the top-k semantic cache can be transparently integrated between the Java application and the JDBC driver (see figure 1.11).

We assume that all queries are projection-selection-join top-k queries in SQL. We will consider neither sub-queries nor aggregations.

Our approach does not consider transactions.

In addition, we assume that data in the database is not changed, i.e., no updates, no deletions and no insertions do occur. Of course, updates can be handled. But they are beyond the scope of this thesis. We outline briefly how updates can be processed by the cache.

Local updates that are passed through the top-k semantic cache are unproblematic. The top-k semantic cache can update its elements and histograms accordingly, and then pass the update on to the server. In the prototype IQCache (see chapter 7), the top-k semantic cache is informed about all \textit{update}, \textit{insert} and \textit{delete} statements that pass through the cache.
Updates from external sources are more complex. There are several possible approaches to this situation. First, the server could maintain an update table for each client. Then, the client can access the update table at a convenient time to pull any available updates. Secondly, the server could maintain one update table with timestamped update information. Doing so, each client can still pull all updates since its last check-up, but the overhead for the server is reduced significantly. Thirdly, the server could send updates to all registered clients and thus push the updates to the clients. This approach would avoid the periodical inquiries of the clients, but may prove problematic if clients are unavailable at times due to network disruptions, which is not unheard of for mobile devices. Finally, the cache could completely forgo updates and rely on the cache replacement policy to keep its data sufficiently current by removing cache elements that contain data that is deemed too old to use. Thereby, the cache can enforce a maximum age for its tuples.

1.6. Contributions

We propose a new semantic cache that naturally supports top-k queries as cache elements.

The top-k semantic cache has the following innovative aspects:

1. Top-k semantic caching: In this thesis, we introduce new techniques to describe the tuples of partially loaded sorted queries. In addition, we also state and prove how to describe the remaining tuples that have not yet been loaded. We present algorithms for the creation of new cache elements that represent top-k queries as well as for calculation of the difference, intersection and union of two of these cache elements. Thereby, we enable the storage of sorted and especially limited
1. Introduction and Motivation

queries, which no other semantic cache can do. Finally, in this thesis, we introduce a novel algorithm for top-k query processing that uses stored top-k query results to compute the answer to the given top-k query.

2. Pipelining of top-k queries: We have developed an innovative algorithm that, using multidimensional histograms, can estimate the lower bounds of query results of sorted queries. With this algorithm, our top-k semantic cache is able to pipeline results of top-k queries. Something, no other semantic cache has been able to do. Thereby, the query execution performance of the top-k semantic cache can be increased.

3. Hybrid solver: We have developed a new SMT solver, a solver for expressions over various domains. Using a novel formula to assess the complexity of expressions, this hybrid solver is able to distinctly outperform all other considered solvers when processing the actual expressions that the prototype IQCache is posing to the solver during various benchmarks. We are the first to apply efficient SMT solvers, which have seen tremendous progress over the last ten years, to semantic caching.

4. Three-valued logic: Because of the incorporation of null values, expressions in SQL use three-valued logic. Other approaches use a simplified view of SQL by assuming only two-valued logic. We do not. In our approach, the cache elements are described by three-valued logic. And the algorithms for cache management and query processing in a top-k semantic cache, which are based on propositions that we have both established and proven in this thesis, operate in three-valued logic.

5. Transparency: The top-k semantic cache provides transparent caching inside a JDBC driver. Thus, it removes the need for an application-specific cache.

6. Datatypes: Other prototypes of semantic caches only support integer variables. In contrast, our prototype IQCache supports various data types like integer, double and varchar. Thereby, the evaluation is not limited to one data type. And the results are more meaningful in relation to a practical application.

1.7. Overview

This thesis consists of three parts. Each of these parts contains three chapters. Part 1 covers the fundamentals of semantic caching. Chapter 1 introduces and motivates semantic caching and especially top-k semantic caching. It presents a motivating running example that will accompany this thesis. It concludes by highlighting the contributions of this thesis. Chapter 2 discusses the related work of semantic caching in great detail.
Chapter 3 deals with the satisfiability of expressions in SQL. First, the chapter introduces the concept of satisfiability over various domains. Then, the chapter examines SMT solver, i.e., solver for expressions over various domains. It studies in-depth which SMT solvers are best suitable for the use case of checking if expressions in SQL are satisfiable. Because SQL uses three-valued logic and supports various domains, the chapter further discusses three-valued logic in SQL and the combination of three-valued logic and satisfiability.

Part II covers the design of a top-k semantic cache. Chapter 4 defines and describes its segments. In addition, the chapter introduces set operations – like difference, intersection and union – on segments. Chapter 5 proposes an algorithm for the estimation of bounds of query results that operates on multidimensional histograms. With this algorithm, the top-k semantic cache is able to pipeline query results of queries that can only be partially answered by the cache. Chapter 6 discusses cache organization and query processing in a top-k semantic cache.

The techniques for top-k semantic caching that are described in this thesis have been implemented in the prototype IQCache. Part III discusses the results obtained in practical experiments using this prototype. Chapter 7 showcases some interesting aspects of the implementation of IQCache. Chapter 8 contains a thorough evaluation of top-k semantic caching using the prototype IQCache and the benchmarks Yahoo! Cloud Serving Benchmark [CST+10], the Twitter benchmark [CHBG10], and some test queries from the MonArch 3 system [SWF11]. Chapter 9 discusses future work like prefetching, context-awareness and the mobile use case. Finally, the last section of this chapter summarizes the core contributions of this thesis.
2. Related Work of Semantic Caching

In this chapter, we will discuss the related work of semantic caching in general. Other related work will be discussed in the relevant sections. For example, during the selection of a suitable solver for a top-k semantic cache, SMT solvers are studied in great detail (see section 3.2.1). And related work on histograms and their typical applications is discussed during the introduction of multidimensional histograms (see section 5.1). Furthermore, related work on cache replacement policies is mentioned when we present the performance of the different replacement strategies in a top-k semantic cache (see section 7.2).

2.1. Semantic Caching of SQL Queries

Semantic caching has first been proposed by S. Dar et al. [DFJ+96] and A. Keller and J. Basu [KB96]. S. Dar et al. propose a semantic cache for simple select-project queries to single relations [DFJ+96]. Their semantic cache does support neither joins nor orderings nor limits. They introduce the basic terminology of semantic caching, including probe query and remainder query. They compare semantic caching with tuple caching and page caching. They conclude that semantic caching generally outperforms tuple and page caching. In addition, they study replacement policies (see section 7.2) and coalescing strategies for semantic caches (see section 4.2 and section 6.2.5).

A. Keller and J. Basu present a semantic cache for select-project-join queries [KB96]. They only allow where conditions with range predicates. They elaborately discuss the challenge of maintaining cache currency due to inserts, updates and deletes. They propose to augment queries to make them more suitable to caching by adding attributes to the select clause or by simplifying where conditions by dropping a part of it. The latter can speed up query processing and cache management, but can also possibly cause a significant increase in result size. This would waste client and server resources as well as strain the network connection. And it can result in the transfer of tuples that may never be used by future queries.

Since then, semantic caching of SQL queries has received great interest and has been widely studied.

P. Godfrey and J. Gryz introduce a logical framework for semantic caching [GG97, GG98]. They consider different types of overlaps and subsumptions of select-project queries and
2. Related Work of Semantic Caching

the semantic cache. But they consider neither sorted nor limited queries. They propose to introduce semantic caching techniques in complex environments like heterogeneous databases or data warehouses to optimize query evaluation. In addition, they briefly discuss the possible application of semantic caching to data security (as sensitive data can be transferred less across the network if it is cached locally), fault tolerance, and approximate query answering. In addition, they mention that semantic caching might enable answer set pipelining, but they do not propose how this could possibly achieved. 

D. Lee and W. Chu present CoWeb \cite{LC99,LC01}, a semantic cache for web sources. Typically, web sources provide less querying possibilities than traditional database systems. Hence, they introduce a query matching algorithm that finds the best matched query based on the capabilities of the web source. CoWeb only supports query with conjunctive predicates. It does not allow joins. They consider neither sorted nor limited queries. In addition, they study semantic locality, i.e., the similarity among queries, and its effect on semantic caches. Concretely, they investigate the influence of the number of attributes in the where conditions and the frequency of occurrence of attributes in the where conditions. Furthermore, they examine the influence of data locality using different query spaces. Given a workload and a database instance, a query space is defined as the set of the tuples of all query results.

Q. Ren et al. introduce a formal semantic caching model \cite{RDK03}. They are the first to formally describe segments and query processing in a semantic cache. But their semantic cache only allows select-project queries of single relations. Furthermore, it is restricted to conjunctive predicates without inequality. They explain coalescence and decomposition, which are needed to avoid redundant data in the semantic cache. In addition, they study the impact of the physical organization of the database like indexing and clustering, the query workload, and the network bandwidth on semantic caching. Using the formal semantic caching model of Q. Ren et al., H. Wan et al. apply semantic caching to data grids and data intensive computing \cite{WHZL04}. Hence, they also only consider select-project queries without joins. They argue that data in a data grid can be classified as structural, i.e., relational, data. Furthermore, they assume that occurring queries have very large results. They have implemented a semantic caching service for a data grid. In their experiments, semantic caching can reduce network traffic.

G. Soundararajan and C. Amza present a template-based semantic cache for select-project-join queries \cite{SA05}. Their semantic cache can only detect overlaps of queries that have the same template (i.e., could be based on the same prepared statement). In addition, it is restricted to queries with conjunctive predicates. But it does allow the aggregations count and max as well as orderings, but no limits. Their semantic cache can process insert statements. Newly inserted tuples are kept in separate tables, one per regular database table. Furthermore, their semantic cache can handle update and delete statements that pass through the semantic cache. They use the TPC-W benchmark \cite{Men02} to evaluate their semantic cache. They can show that their semantic cache is able to achieve a significant performance improvement in their experiments.
2.1. Semantic Caching of SQL Queries

J. Cai et al. introduce semantic query processing for aggregate select-project-join queries [CJYZ05]. Their approach supports the operator group by and the aggregations min, max, count, and sum. They have implemented their semantic cache as part of their StarTP database system. They claim that first evaluations show very promising results, but their paper does not contain any experiments.

M. Bashir and M. Qadir propose a 4-level hierarchical indexing scheme to improve query processing inside a semantic cache [BQ06a]. The scheme indexes cached queries using the hierarchical information provided by the SQL query itself (database, table, attribute, and predicate).

B. Jónsson et al. revisit semantic caching of select-project queries over single relations. They present a detailed performance analysis of semantic caching using various workloads [JAT06]. For simple workloads, their performance study shows that semantic caching produces low overhead, is insensitive to clustering, unburdens the network, and can answer queries without contacting the server. For more complex workloads, they have uncovered that semantic caching does strain the server. Because complex remainder conditions in the remainder queries require more effort from the server. But even for complex workloads, their semantic cache has been able to use the network efficiently and successfully reduce query response times. They conclude that semantic caching can be applied to various workloads and has a wide range of applications.

M. Abbas and M. Qadir analyze the formal semantic cache model of Q. Ren et al.. They discuss and exemplify possible weaknesses of the approach [AQ09].

M. Ahmad et al. enhance the 4-level hierarchical indexing scheme of M. Bashir and M. Qadir. They claim that their scheme is able to reduce the number of segments that have to be considered during query processing [AAQA10]. We also use a scheme to partition our cache. However, in our top-k semantic cache, we use equivalence classes based on the existing joins and a fast, heuristic join detection algorithm to reduce the number of segments that have to be considered for a posed query (see section 6.1).

N. Ryeng et al. apply semantic caching to a distributed database system [RHN11]. Thereby, the caches of nodes of the distributed database system are not limited to base tables, but can contain intermediate results. They study different LRU-based replacement policies. They evaluate their system with the TPC-H benchmark [Tra]. Using semantic caches, they can reduce query execution time by 40 to 50 percent. In our evaluation, we have achieved a similar result (see chapter 8). They conclude that semantic caching in distributed database systems enables scaling the system without excessive network traffic.

There are surveys on semantic caching by B. Jónsson et al. [JAT06], M. Bashir and M. Qadir [BQ06b], M. Ahmad et al. [AQS08], and P. Kumar et al. [KDV13].

Altogether, none of the surveyed semantic caches can store limited queries. Some existing approaches can process limited queries. But they do so by removing the limit. They pose the query without the limit to the database and then cache the result.
of that query. This approach might be utilized successfully for server-side caching \cite{APT03, LNX08}. But it is not feasible for client-side caching, because it transfers too many irrelevant tuples through the bottleneck between server and client. In contrast, our top-k semantic cache that is presented in this thesis is able to store limited queries.

In addition, none of the surveyed semantic caches can pipeline partial matches of queries with orderings. To directly answer a sorted query from the semantic cache, existing approaches need the cache to contain the complete answer. In contrast, our top-k semantic cache is able to pipeline the query result if only a partial match is found in the cache. To achieve this performance enhancement, it uses an algorithm that operates on multidimensional histograms (see section 5.3).

Furthermore, in the last ten years, the research community has made tremendous progress in the development of efficient solvers for logical expressions over various domains, i.e., SMT solvers. This research should spark renewed interest in semantic caching. We are the first to apply SMT solvers to semantic caching. We have built a hybrid solver that is especially tuned to solve the expressions that occur during the operation of a semantic cache (see section 3.2.2).

### 2.2. Semantic Caching of Non-SQL Queries

Semantic caching has also been employed for a multitude of non-SQL applications. For the sake of completeness, we will briefly survey the application of semantic caching to XML databases, web queries, information retrieval, and spatial data.

#### 2.2.1. XML Databases

Semantic caching has also been applied to XML databases. 

* L. Chen et al. present XCache \cite{CRW02}, a semantic cache for XML queries. Their algorithm for query containment is based on an algorithm for subtyping in XML \cite{HVP05}. 

* G. Li et al. propose SCEND \cite{LFT06}, a semantic cache that contains materialized XPath views. In an XML database, their semantic cache enables faster XML query processing. To enhance the number of queries that can be answered using the semantic cache, they have developed an algorithm to decompose complex queries into simpler ones.

* M. Sumalatha et al. also present a semantic cache for XML queries \cite{SVK07a, SVK07b}. Their semantic cache similarly splits complex queries into simple sub-queries. Furthermore, it organizes the cached elements in a tree structure.
2.3. Other Related Caching Techniques

2.2.2. Web Queries and Information Retrieval

B. Chidlovskii and U. Borghoff propose a semantic cache for conjunctive boolean keyword-based web queries [CB00]. Disjunctions are not directly supported. Queries that contain disjunctions are transformed into a disjunctive normal form and split into conjunctive sub-queries. Their semantic cache can exploit subsumptions and overlaps of queries.

Q. Luo et al. have studied form-based proxy-caching of top-k conjunctive keyword-based web queries for database-backed web servers [LN01, Luo05, LNX08]. Because web searches with form-based interfaces enforce a common structure on the queries, their semantic caching scheme can use a template-based approach. The top-k operator is removed from a query before the query is sent to the web server. To support top-k queries, they have implemented the top-k operator in their semantic cache, which will apply it to the full result before it is passed to the user.

E. Benson et al. present Sync Kit [BMKM10], a client-side in-browser cache inspired by semantic caching that uses the persistent database API of HTML5.

A. Peters and A. Heuer argue that semantic caching of boolean web queries can be used to reduce data loss in ad-hoc networks [PH11]. They present BlueS, a service-based framework for data accessibility and management in ad-hoc environments.

B. Cambazoglu et al. apply semantic caching techniques to text retrieval [CAOU12]. They employ the result cache of a search engine. They propose query processing strategies to answer previously unseen user queries using the result cache. In their experiments, their approach could retrieve at least two relevant results for more than 75 percent of the queries.

2.2.3. Spatial Data

B. Zheng et al. present a semantic cache that accelerates mobile nearest-neighbor search [ZLL04]. They define the semantic region of a spatial object as its Voronoi cell.

H. Hu et al. discuss semantic caching of tree nodes in an R-tree [HXW05]. They examine how to process spatial queries on the cached tree nodes.

J. Thompson introduce semantic caching of shortest paths for location-based services [TYJ12]. Each sub-path of a shortest path is a shortest path. Hence, if two paths share two nodes, they will share a sub-path, i.e., the paths will overlap. Their semantic cache exploits this observation.

2.3. Other Related Caching Techniques

The following caching techniques are related to semantic caching, because they also present algorithms that make use of subsumptions of queries. But they are not effective
strategies for client-side caching in an environment with low throughput and high latency, since, for instance, they do not support partial answering, only allow for conjunctive queries, or focus on mid-tier or server-side caching, respectively.

2. Related Work of Semantic Caching

2.3.1. Logical Caching

K. Amiri et al. present DBProxy [APTP03, APT03], a logical cache for select-project-join queries. DBProxy does not support partial matches. It can only answer queries that are completely contained within the cache. Hence, it is not a semantic cache. In addition, their cache only supports where conditions in disjunctive normal form. It does permit orderings and limits. But DBProxy cannot store results of limited queries. It processes limited queries by removing the limit. It retrieves the result of the query without the limit, and caches the result of that query. To improve the performance of query containment checking, their cache can create equivalence classes of queries that match the same template. They have evaluated DBProxy using the TPC-W benchmark [Men02]. In their experiments, their cache was able to significantly accelerate workload of the benchmark.

2.3.2. Table Caching

A table cache contains tables or sub-tables instead of query results. Commonly, the cached tables are statically defined by the user. During query processing, a table cache may utilize techniques that are also similarly used in semantic caches. The TimesTen Team presents TimesTen, an in-memory mid-tier cache for applications with a data-intensive workload [The02]. The user has to choose the cached tables using a browser-based tool. The cache supports the ACID properties. It unburdens the central database by offloading data processing work. TimesTen has been acquired by the Oracle Corporation in 2005. Data-intensive applications that use an Oracle database can now utilize TimesTen as an in-memory cache database in front of the Oracle Database. Luo et al. introduce transparent caching of full tables at mid-tier level to the IBM DB2 database [LKM+02]. Initially, the selection of the cached tables had to be conducted by the user. Their cache, called DBCache, supports distributed query processing, i.e., the cache uses query plans that may involve both the cache and the remote server. Altinel et al. have enhanced DBCache to also be able to cache parts of tables and dynamically choose the cached tables [ABK+03]. Larson et al. present MTCache, a caching solution similar to DBCache for the Microsoft SQL Server [LGZ03]. T. Härder and A. Bühmann analyze the technical challenges of using fully-fledged database management systems as mid-tier caches [HB08]. Particularly, they discuss TimesTen and DBCache in their work. W. Wang et al. present EasyCache [WLJ+14], which uses Hazelcast, an in-memory data grid, for distributed data storage. The cache transforms a table of the database into a
2.3. Other Related Caching Techniques

Hazelcast map. Each row of the table is converted into a key-value pair. The primary key of the table is used as the key in the Hazelcast map. EasyCache itself is implemented inside a JDBC driver. EasyCache accepts select-project-join queries and translates them into key-value operations on Hazelcast. Currently, EasyCache supports neither orderings nor limits. Nevertheless, in their evaluation using the TPC-W benchmark [Men02], EasyCache was able to significantly improve response times.

2.3.3. Answering Queries Using Views

Answering queries using views is engaged in how to find a rewriting of a given query using materialized views. The rewriting has to be complete, i.e., the query has to be completely answered by the set of views. In addition, the rewriting should preferably be minimal.

A. Levy et al. have been the first to thoroughly study answering queries using materialized views [LMS95]. They only consider queries with conjunctive predicates and materialized views based on conjunctive queries. In their work, they analyze the complexity of the difficulty of finding minimal or complete rewritings, respectively. They prove that both of the problems are NP-complete. In addition, A. Levy et al. introduce an algorithm for rewritings of queries with conjunctive predicates, the bucket algorithm [LRO96].

X. Qian also presents an exponential-time algorithm that finds all possible query rewritings of a given conjunctive query. In addition, he provides a polynomial-time algorithm, the inverse-rules algorithm, for the special case of conjunctive queries without cyclical joins [Qia96]. M. Duschka and M. Genesereth have implemented the inverse-rules algorithm in their database system Infomaster [DG97].

R. Pottinger and A. Halevy introduce MiniCon [PH01], a scalable algorithm that finds rewritings of a conjunctive query using views. It is able to outperform both the bucket algorithm and the inverse-rules algorithm.

The algorithms for answering queries using views have been applied to caching. For example, I. Brunkhorst and H. Dhraief introduce answering queries using cached views to P2P networks [BD07]. Their Edutella project aims to build a scalable, schema-based infrastructure for the Semantic Web. They propose to add caching on end-user nodes. They use the MiniCon algorithm. Consequently, their cache supports select-project-join queries with conjunctive predicates. But the proposed cache does not support partial containment.

2.3.4. Caching of Top-k Queries with Linear Score Functions

Much research in the processing of top-k queries with linear score functions like Fagin’s algorithm [Fag96], the Threshold algorithm [FLN01], the No Random Access algorithm [FLN01], and the Stream-Combine algorithm [GBK01] has sparked interest in answering top-k queries with linear score functions using views.
2. Related Work of Semantic Caching

First, V. Hristidis et al. present the PREFER algorithm [HKP01]. This algorithm selects only one materialized view to answer a query. It cannot use multiple views. G. Das et al. propose the LPTA algorithm [DGKT06]. This algorithm can utilize multiple materialized views to answer a query. But it needs either each materialized view to contain a complete ranking of all tuples, or available attribute base views, which are complete rankings of all tuples according to the values of the attribute. M. Xie et al. introduce several improvements to the LPTA algorithm with their algorithm LPTA+ [XLW13]. Their enhancements allow the algorithm LPTA+ to find certain answers even when a complete ranking of all tuples in the view or attribute base views are not available. In addition, they present IV-Search, a new algorithm for answering top-k queries with linear score functions using views that utilizes an inverted view index. This index stores the contents of all materialized views in a central data structure in memory. Thereby, it significantly improves the performance of view selection.

All these approaches only support queries with conjunctive range predicates. They only allow linear score functions. The reason for these restrictions is that these algorithms use linear programming, e.g., the SIMPLEX algorithm [Dan98], during query processing. In contrast, in our top-k semantic cache, we use logical descriptions and SMT solver. Hence, we support non-linear score functions, especially lexicographical orderings. In addition, our approach is not limited to conjunctive predicates.
3. Satisfiability

In the early seventies, Stephen A. Cook and Leonid Levin independently showed that any problem in NP can be reduced with a polynomial time many-one reduction to the boolean satisfiability problem (SAT), i.e., the problem of determining whether a boolean formula is satisfiable [Coo71, Lev73]. Today, their ground-breaking result is known as the Cook-Levin theorem.

The proof showed that there exists a subset of NP that all problems of NP can be reduced to with a polynomial time many-one reduction. This subset is representative for the difficulty of NP. It is called NP-complete.

The proof sparked much research into NP-complete problems. In 1972, Richard M. Karp published 21 NP-complete problems [Kar72]. And in 1979, Michael R. Garey and David S. Johnson presented a book with more than 300 NP-complete problems [GJ79]. New NP-complete problems are still being discovered every day.

Because all problems in NP can be reduced to the boolean satisfiability problem (SAT), a solver for SAT can in principle solve all problems of NP. And as it was shown that many practically relevant problems are NP-complete, much research in SAT solvers has been conducted. By modern SAT solvers, even though SAT is NP-complete in general, many practical instances can be solved efficiently using heuristic methods.

In recent years, the extension of SAT from the boolean domain to various domains, e.g., the integer, real number or bit vector domain, has gained much attention. This extension is called satisfiability modulo theories (SMT). SMT solvers are now available that can solve these extensions of SAT very efficiently, even though, they typically remain NP-complete.

While satisfiability modulo theories (SMT) is based on two-valued propositional or first-order logic, SQL uses a three-valued logic based on the common subset of Kleene logic and Łukasiewicz logic [Kle50, Luk70, Kle03]. Fortunately, SMT solvers can be used for SQL expressions after applying a transformation function to the three-valued logical expressions of SQL [BG04, BG05].

3.1. Satisfiability and Domains

The boolean satisfiability problem (SAT) only considers variables from the boolean domain (\(\mathbb{B}\)). We will extend the satisfiability problems to various domains including
3. Satisfiability

Integers (\(\mathbb{Z}\)), real numbers (\(\mathbb{R}\)) and varchars (\(S_B(n)\), see definition 3). An overview of the domains can be seen in table 3.1.

<table>
<thead>
<tr>
<th>domain</th>
<th>domain description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathbb{B})</td>
<td>booleans</td>
</tr>
<tr>
<td>(\mathbb{Z})</td>
<td>integers</td>
</tr>
<tr>
<td>(\mathbb{R})</td>
<td>real numbers</td>
</tr>
<tr>
<td>(S_B(n))</td>
<td>varchars over alphabet (B) with maximum length (n) (see definition 3)</td>
</tr>
</tbody>
</table>

Table 3.1.: Overview of domains

3.1.1. Variables and Domains

To create expressions using variables from various domains, we need to be able to define a variable from a certain domain.

**Definition 1 (Variable)** Let \(M\) be a set. Then, we define \(\text{var}(M)\) as the set of all variables with domain \(M\).

On the other hand, we must be able to get the domain of a variable.

**Definition 2 (Domain function)** The domain function \(\text{dom}\) returns the domain of a variable. If \(x \in \text{var}(M)\) is a variable, \(\text{dom}(x) = M\).

**Example 1 (Variable and Domain)** Let \(x\) be a variable with \(x \in \text{var}(\mathbb{Z})\). Then, \(x\) is an integer variable, i.e., a variable from the integer domain. Therefore, \(\text{dom}(x) = \mathbb{Z}\).

3.1.2. The Varchar Domain

While \(\mathbb{B}\), \(\mathbb{Z}\) and \(\mathbb{R}\) are well-known, well-defined, widely used sets, we do have to define the varchar domain. Along the way, we will also define the string domain, which is very similar to the varchar domain, but does not have a maximum length restriction.

**Definition 3 (String Domain, Varchar Domain)** The *string domain* \(S_B\) contains all character sequences that uses characters from the alphabet \(B\).

\[
S_B \overset{\text{def}}{=} \{b_1 \circ \ldots \circ b_i \mid i \in \mathbb{N}_0 \land \forall j \in \{1, \ldots, i\} : b_j \in B\}
\]

The *varchar domain* \(S_B(n)\) contains all character sequences that uses characters from the alphabet \(B\) that have a length of at most \(n\) letters:

\[
S_B(n) \overset{\text{def}}{=} \{b_1 \circ \ldots \circ b_i \mid i \in \{0, \ldots, n\} \land \forall j \in \{1, \ldots, i\} : b_j \in B\}
\]

Therefore, every varchar domain can be seen as a string domain with a maximum length restriction.
3.1. Satisfiability and Domains

Example 2 (Varchar Domain) If we restrict the alphabet to 0 and 1 and the maximum length to two, the following seven different varchars are possible:

\[ S_{\{0,1\}}(2) = \{ \epsilon, '0', '1', '00', '01', '10', '11' \} \]

Proposition 1 (Size of Varchar Domain) The number of varchars in a varchar domain grows rapidly with an increasing alphabet and a greater maximum length. It can be described by the following formula:

\[
|S_B(n)| = \begin{cases} 
1 & |B| = 0 \\
|B| + 1 & |B| = 1 \\
\frac{|B|^{n+1} - 1}{|B| - 1} & |B| \geq 2
\end{cases}
\]

Proof. For \(|B| = 0\), only the empty sequence \(\epsilon\) is possible. Hence, \(|S_B(n)| = 1\). For \(|B| = 1\), we can repeat the single character between 0 and \(n\) times. Therefore \(n + 1\) different sequences can be built. Finally, for \(|B| \geq 2\), we can construct 1 empty sequence, \(|B|\) sequences of length 1, \(|B|^2\) sequences of length 2 and so on. Thereby, \(|S_B(n)| = 1 + |B| + |B|2 + \ldots + |B|^n = \frac{|B|^{n+1} - 1}{|B| - 1}\). □

Example 3 (Size of Varchar Domain) For an alphabet of 128 different characters (i.e., 1 byte) and for a maximum length of 30, the varchar domain contains the astronomical size of more than \(10^{63}\) different varchars (\(\frac{128^{30} - 1}{127} > 10^{63}\)).

Strings and varchars are typically ordered lexicographically.

Definition 4 (Lexicographical Order on Varchars) Let \(B\) be an ordered alphabet and let \(s = a_1 \ldots a_m \in S_B(m)\) and \(t = b_1 \ldots b_n \in S_B(n)\) be varchars. The lexicographical order \(<_S\) on the varchars \(s\) and \(t\) is defined as follows:

\[ s <_S t \iff (m_s < n_t \land \forall j \in \{1, \ldots, m_s\} : a_j = b_j) \lor (\exists i \in \{1, \ldots, \min\{m_s, n_t\} \}: (a_i <_B b_i \land \forall j \in \{1, \ldots, i - 1\} : a_j = b_j)) \]

Example 4 (Lexicographical Order on Varchars) The varchars of the varchar domain from example 2, i.e., \(S_{\{0,1\}}(2)\), the varchar domain with the alphabet \(\{0, 1\}\) and the maximum length two, are lexicographically ordered as follows:

\[ \epsilon <_S '0' <_S '00' <_S '01' <_S '1' <_S '10' <_S '11' \]
3. Satisfiability

3.1.3. From SAT to SMT

First, we investigate a typical boolean satisfiability problem.

**Example 5 (Boolean satisfiability problem)** Given the boolean variables \( a, b, c \in \text{var(B)} \), consider the following boolean expression in conjunctive normal form:

\[
(a \lor b) \land (\neg b \lor \neg c) \land (\neg a \lor c) \land (\neg a \lor b \lor \neg c)
\]

We want to figure out if the expression is satisfiable. To do this, we need to find an assignment of truth values to the boolean variables \( a, b \) and \( c \) that satisfies the expression. In fact, such an assignment does exist, namely, we can assign false to \( a \), true to \( b \) and false to \( c \).

Satisfiability modulo theories (SMT) is an extension of SAT from the boolean domain to various domains, e.g., the integer, real number or varchar domain. For instance, in satisfiability modulo theories, boolean expressions can be enriched with linear constraints.

**Example 6 (Satisfiability modulo theories)** Given the varchar variables \( x, y, z \in \text{var(S}_{\{0,1\}(2)}) \), consider the following conjunctive expression with linear constraints:

\[
('01' \leq x) \land (x < y) \land (y < z) \land (z \leq '1')
\]

Again, we want to work out if the given expression is satisfiable. Hence, we have to find a legal assignment of values to the varchar variables \( x, y \) and \( z \) that satisfies the expression.

Actually, there is no such assignment. The variable \( x \) must be assigned at least '01' as \( x \geq '01' \). Thus, we assign '01' to \( x \). The variable \( z \) has to be assigned at most '1', because \( z \leq '1' \). Therefore, we assign '1' to \( z \). The value \( \beta_y \) that we assign to variable \( y \) must be greater than the value that we have assigned to \( x \) and smaller than the value that we have assigned to \( z \), i.e., \( '01' < \beta_y < '1' \). But it must also be a varchar with maximum length of two, i.e., \( \beta_y \in S_{\{0,1\}(2)} \). As seen in example 4, such a value does not exist.

Interestingly, the same expression does become satisfiable if the varchar variable \( y \) has a greater maximum length, e.g., \( y \in \text{var(S}_{\{0,1\}(3)}) \). Because \( '01' < '010' < '011' < '1' \) holds, we can assign '010' or '011' to \( y \). And the assignment of '01' to \( x \), '010' to \( y \) and '1' to \( z \) satisfies the expression above.

Accordingly, we can conclude that the maximum length of the utilized varchar variables has an impact on the satisfiability of expressions containing those varchar variables. Hence, an SMT solver will have to take the domains of the used variables – e.g., the integer, real number or varchar domain – into consideration when it determines whether a given expression is satisfiable.
3.2. SMT Solver

SMT solvers determine the satisfiability of expressions over various domains. In this section, we will discuss the selection of a suitable SMT solver for a top-k semantic cache. In addition, we will introduce a new hybrid solver, which is able to outperform all other considered solvers.

3.2.1. Solver Selection

We need an SMT solver to check if certain SQL expressions are satisfiable. For this reason, we have a couple of requirements that the SMT solver we eventually select for the top-k semantic cache must meet.

First of all, the SMT solver should be current with an active team behind it. It must not return false results for satisfiability checks, i.e., it must operate correctly. It must have a free license. It must be well documented. Available source code would be a plus, because, therewith, we could disable unused features or add new ones. It must be available for Windows 7 (64 bit) as this is our development platform. And it should support either the input format SMT-LIB 1.2 or SMT-LIB 2.0 [BST10]. A Java integration, e.g., in form of an API, would be an advantage as our prototype IQCache is implemented in Java.

SMT Solvers usually support different logics. These describe its functional range. They have been standardized by the SMT-LIB initiative, "an international initiative aimed at facilitating research and development in Satisfiability Modulo Theories (SMT)", e.g., SMT solvers [BST10].

Databases, on the other hand, usually support many data types. As a start, we want to at least support integers, doubles and varchars. That is why the selected SMT solver must be able to check if expressions with these data types are satisfiable. Hence, the SMT solver must support the three following logics:

- **QF_LIA** (unquantified linear integer arithmetic), i.e., boolean combinations of inequations between linear polynomials over integer variables [BST10]
- **QF_LRA** (unquantified linear real arithmetic), i.e., boolean combinations of inequations between linear polynomials over real variables [BST10]
- **QF_BV** (closed quantifier-free formulas over the theory of fixed-size bitvectors) [BST10]

Another important aspect is the performance of the SMT solver. In fact, there is an annual competition for SMT solvers called the SMT-COMP which was first held in 2005 [SMT05, SMT06, SMT07, SMT08, SMT09, SMT10, SMT11, SMT12, SMT14].

1Competitions: [SMT05, SMT06, SMT07, SMT08, SMT09, SMT10, SMT11, SMT12, SMT14]
3. Satisfiability

The participating SMT solvers are compared in various categories by their total execution time over many benchmarks, which mostly consist of large and complex expressions.

In contrast, in a top-k semantic cache, an SMT solver has a very different workload. It must check a lot of very comparatively small expressions in a very short time frame. Therefore, the overhead of the SMT solvers becomes much more important. As a consequence, the results of the SMT-COMPs can only be taken as an indicator. That is why we need a detailed evaluation of SMT solvers with regard to their usage in a top-k semantic cache. We have overseen several theses that evaluated many SMT solvers with regard to this [Hüb10, Jac11, DT14].

In his 2010 bachelor's thesis “Semantic Caching with Linear Constraints” [Hüb10], Pius Hübl surveyed the solvers Choco [FP], Cream [Tam], Dragonbreath Engine [Nar], ILOG J-Solver [BMb], JaCoP [KS], JACK [AFK+AKSS02], JCL [TWF97], Koalog [Koa], Open3DCFD [Bus], SUGAR [ITKB09], Temporal Constraint Solver [Gre], and Yices [DM06].

After careful consideration, for a detailed evaluation, he chose the Choco solver, which is written in Java and provides a Java API, and the Yices solver, which he integrated into the evaluation environment using the Java Native Interface (JNI). Thereafter, he elaborately compared the Choco solver, the Yices solver and the DNF solver. The latter is a fast SMT solver for simple expressions and uses techniques described in [GSW96] and [SB00]. We have implemented the DNF solver as part of our prototype IQCache, i.e., the prototype of a top-k semantic cache.

The results of the evaluation were clear-cut. The Yices solver, despite its integration via JNI, was able to beat the Choco solver distinctly. And the DNF solver proved to be extremely fast and unmatched for simple expressions. But, as expected, it could not cope well with more complex expressions.

In consequence of these results, we have integrated the Yices solver into the prototype IQCache.

Jacobs' 2011 bachelor's thesis “Evaluation of SMT Solvers for Usage in a Semantic Cache” [Jac11] considered the SMT solvers AProVENIA [BDE+14], Beaver [JLS09], CVC3 [BT07], DPT [GG], MiniSMT [ZM10], SatEEn [KJS09], SONOLAR [Lap10], Spear [BH08], veriT [BCbODF09], Yices [DM06], Yices 2 [Dut14], and Z3 [DMB08].

After surveying and researching the solvers, he selected CVC3, Yices 2 and Z3 for comparison with themselves and the already integrated solvers, Yices and DNF. He chose not to use the Java Native Interface and integrated CVC3, Yices 2 and Z3 as a process into the evaluation environment.

The evaluation showed that the process integration produces too much overhead. An integration via JNI is much preferable. He could demonstrate that Yices 2 and Z3 have a much better performance than CVC3 for typical top-k semantic cache workloads. In addition, the fixed cost, i.e., the overhead for a single satisfiability check, of Z3 proved to be consistently higher than the one of Yices or Yices 2.
3.2. SMT Solver

In her 2014 bachelor’s thesis “Evaluation of SMT Solvers and SMT-Lib 2.0 for Usage in a Semantic Cache” [DT14], Mailan Dinh-Thanh studied the solvers Alt-Ergo [CIM13], Barcelogic [BNO+08], Beaver [JLS09], Boolector [BB09], CVC4, [BCD+11], MathSAT 5 [CGSS13], MiniSMT [ZM10], Mistral [DDMA12], Open SMT [BPST10], SMT Interpol [CHN12], SONOLAR [Lap10], STP [GD07], veriT [BCBGdOF09], Yices [DM06], Yices 2 [Dut14], and Z3 [DMB08]. After thoroughly considering all solvers, she picked MathSAT 5 and Z3. She integrated them into the prototype IQCache using the provided Java APIs, which utilize the Java Native Interface.

She used the actual expressions of the prototype IQCache during the Yahoo! Cloud Serving Benchmark [CST+10], the Twitter benchmark [CHBG10], and while executing real queries of the MonArch 3 system [SWF11] to comprehensively and extensively evaluate the usage of MathSAT 5, Z3 and already integrated Yices in a top-k semantic cache.

In the evaluation, Yices performed far better for simpler expressions than MathSAT 5. But the latter was considerably faster for more complex expressions. There was no clear-cut winner. Only Z3 could be discarded, because, again, its fixed cost made it the clear looser of the comparison.

The observation that for differently complex expressions different SMT solvers should be chosen to achieve optimal overall performance has led to the development of a hybrid solver.

3.2.2. Hybrid Solver

The evaluations of several theses indicated that different solvers are best for differently complex expressions [Hüb10, Jac11, DT14]. The DNF solver is best used for very simple expressions. The Yices solver is very effective for simple to moderately complex expressions. The MathSAT 5 solver should be used for complex expressions. To choose the best of these solvers for a given expression, we need a quick way to select the most suitable, i.e., the fastest solver with regard to a given expression. Therefore, we need a way to quickly estimate the complexity of an expression. But prior to that, we have to find a reliable measure that can express perceived complexity with hard numbers.

We investigated many dimensions like the number of conjunctions, the number of disjunctions, the number of negations, the length of the examined expression and combinations of the aforementioned ones. None of these proved to be an accurate measure to determine the most suitable SMT solver.

Finally, we did find a measure that works extremely, almost astonishingly, well at selecting the fastest SMT solver for the current expression: the length of the straightforward disjunctive normal form of the negation-free conversion of the original expression.

The removal of all negations from an expression is done by pushing all negations to the leaves of the expression using the de Morgan’s laws. At the leaves, any negation
3. Satisfiability

can be removed by negating the leaf expression, e.g., \(\neg(x \leq y + 28) \equiv (x > y + 28)\) [SB00]. Clearly, this conversion only takes linear time with respect to the length of the expression, i.e., for an expression \(\hat{c}\), it takes \(O(|\hat{c}|)\).

We developed the algorithm dnfLength (see algorithm 1) to calculate the length of the straightforward disjunctive normal form of a given negation-free expression.

**Algorithm 1** dnfLength(\(\hat{c}\))

**Input:** negation-free expression \(\hat{c}\) (consisting of conjunctions (\(\land\)), disjunctions (\(\lor\)) and leaf expressions, but no negations (\(\neg\)))

**Output:** total number of disjunctions \(o\) and total number of conjunctions \(a\) in the disjunctive normal form of negation-free expression \(\hat{c}\) as pair \((o, a)\)

1: if \(\hat{c}\) is leaf expression then
2: return \((0, 0)\)
3: else if \(\hat{c} = \bigvee_{j=1}^{n} \hat{c}_j\) then
4: for each \(j \in \{1, \ldots, n\}\) do
5: \((o_j, a_j) := \text{dnfLength}(\hat{c}_j)\)
6: end for
7: \(o := \left(\sum_{j=1}^{n} o_j\right) + (n - 1)\)
8: \(a := \sum_{j=1}^{n} a_j\)
9: return \((o, a)\)
10: else if \(\hat{c} = \bigwedge_{j=1}^{n} \hat{c}_j\) then
11: for each \(j \in \{1, \ldots, n\}\) do
12: \((o_j, a_j) := \text{dnfLength}(\hat{c}_j)\)
13: end for
14: \(o := \left(\prod_{j=1}^{n} (o_j + 1)\right) - 1\)
15: \(a := \left(\prod_{j=1}^{n} (o_j + 1)\right) \cdot \left(\sum_{j=1}^{n} \frac{a_j}{o_j + 1}\right) + (n - 1)\)
16: return \((o, a)\)
17: end if
18: return

The idea of the algorithm is to characterize a disjunctive normal form by its total number of disjunctions and its total number of conjunctions. For example, the disjunctive normal form \((\hat{b}_1) \lor (\hat{b}_2 \land \hat{b}_3) \lor (\hat{b}_4 \land \hat{b}_5 \land \hat{b}_6)\) is described by the pair \((2, 3)\). The algorithm works recursively. It has three cases.
The termination condition of the recursion occurs if the current expression $\hat{e}$ is a leaf expression. Then, the algorithm will just return $(0, 0)$, because leaf expressions contain neither disjunctions nor conjunctions by definition.

If the current expression $\hat{e}$ is a disjunction of sub-expressions $\hat{e}_1, \ldots, \hat{e}_n$, first of all, we recursively calculate the number of disjunctions and conjunctions in the straightforward disjunctive normal form of these sub-expressions. Therewith, we can calculate the number of disjunctions and conjunctions $(o, a)$ of expression $\hat{e}$ as follows:

$$(o, a) = \left( \sum_{j=1}^{n} o_j + (n - 1) , \sum_{j=1}^{n} a_j \right)$$

On the other hand, if the current expression $\hat{e}$ is a conjunction of sub-expressions $\hat{e}_1, \ldots, \hat{e}_n$, once again, we recursively calculate the number of disjunctions and conjunctions in the straightforward disjunctive normal form of these sub-expressions. Using these recursively computed results, the algorithm calculates the number of disjunctions and conjunctions $(o, a)$ of expression $\hat{e}$ as follows:

$$(o, a) = \left( \prod_{j=1}^{n} (a_j + 1) - 1 , \prod_{j=1}^{n} (o_j + 1) \cdot \left( \sum_{j=1}^{n} \frac{a_j}{o_j + 1} \right) + (n - 1) \right)$$

As a matter of fact, the algorithm dnfLength does not only estimate but does correctly calculate the number of disjunctions and conjunctions in the straightforward disjunctive normal of the given negation-free expression.

**Proposition 2 (Correctness of algorithm dnfLength)**

The algorithm dnfLength (see algorithm 1) correctly calculates the total number of disjunctions and the total number of conjunctions in the straightforward disjunctive normal form of the given negation-free expression.

**Proof.** The correctness of algorithm dnfLength (see algorithm 1) can be proven by structural induction.

**Base case:** A simple expression contains neither disjunctions nor conjunctions. Therefore, the return value $(0, 0)$ is correct.

**Case \( \lor \):** Let the disjunction consist of $n$ disjunctive normal forms. Consider the $j$th one. Let the $j$th disjunctive normal form consist of $m_j$ conjunctive parts. To disjunctively connect them, we need $m_j - 1$ disjunctions. Therefore, the number of disjunctions $o_j$ of the $j$th disjunctive normal form is

$$o_j = m_j - 1.$$
3. Satisfiability

Let $a_{ji}$ for $i \in \{1, \ldots, m_j\}$ be the number of conjunctions in the $i$th conjunctive part of the $j$th disjunctive normal form:

$$\bigvee_{j=1}^{n} \left( \bigwedge a_{j1} \lor \cdots \lor \bigwedge a_{jm_j} \right)$$

Then, the total number of conjunctions of the $j$th disjunctive normal form can be calculated as follows:

$$a_j = \sum_{i=1}^{m_j} a_{ji}$$

With these intermediate results in mind, we consider the current expression. We need $n - 1$ disjunctions to connect the $n$ disjunctive normal forms and thereby create one big disjunctive normal form. The resulting disjunctive normal form still contains all disjunctions of all sub-expressions. Hence, the total number of disjunctions of the current expression is

$$o = \begin{cases} (1) & \left( \sum_{j=1}^{n} (m_j - 1) \right) + (n - 1) \\ (2) & \left( \sum_{j=1}^{n} a_j \right) + (n - 1) \end{cases}$$

Since we can just disjunctively connect the $n$ disjunctive normal forms to create one big disjunctive normal form, the total number of conjunctions in all parts:

$$a = \begin{cases} (1) & \sum_{j=1}^{n} \sum_{i=1}^{m_j} a_{ji} \\ (2) & \sum_{j=1}^{n} a_j \end{cases}$$

In conclusion, the total number of disjunctions and conjunctions of the current expression can be calculated as follows:

$$(o, a) = \left( \left( \sum_{j=1}^{n} a_j \right) + (n - 1) , \sum_{j=1}^{n} a_j \right)$$

Hence, we actually only need to know the number of disjunctions and conjunctions of each sub-expression to accurately calculate the total number of disjunctions and conjunctions of the current expression.
3.2. SMT Solver

Case $\land$: Let the conjunction consist of $n$ disjunctive normal forms. Consider the $j$th one. As in the previous case, let the $j$th disjunctive normal form consist of $m_j$ conjunctive parts. And let $a_{ij}$, for $i \in \{1, \ldots, m_j\}$ be the number of conjunctions in the $i$th conjunctive part of the $j$th disjunctive normal form:

$$\bigwedge_{j=1}^{n} \left( a_{j1} \lor \ldots \lor a_{jm_j} \right)$$

To create a disjunctive normal form for the current expression, we use the distributivity of conjunction and disjunction to expand the expression. Since the $j$th disjunctive normal form consists of $m_j$ conjunctive parts, combinatorics dictates that the expansion contains $\prod_{j=1}^{n} m_j$ different conjunctive parts. To disjunctively connect these conjunctive parts, we need $(\prod_{j=1}^{n} m_j) - 1$ disjunctions:

$$o = (\prod_{j=1}^{n} m_j) - 1$$

By distributivity, a conjunctive part of the expansion is the conjunction of $n$ conjunctive parts, one from each of the $n$ disjunctive normal forms. Let $i_j$ be the chosen conjunctive part of the $j$th disjunctive normal form. Then, by definition, this conjunctive part consists of $a_{ij_j}$ conjunctions. In addition, we need $n - 1$ conjunctions to connect the individual parts. Therefore, the number of conjunctions of the considered conjunctive part of the expanded expression can be calculated as follows:

$$\left( \sum_{j=1}^{n} a_{ij_j} \right) + (n - 1)$$

The total number of conjunctions in the current expression is the sum of the number of conjunctions of all combinations of the conjunctive parts of the $n$ disjunctive normal forms, which can be transformed as follows:

$$a = \sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} \left( \sum_{j=1}^{n} a_{ij_j} \right) + (n - 1)$$

$$= \left( \prod_{j=1}^{n} m_j \right) \cdot (n - 1) + \sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} a_{ij_j}$$

$$= \left( \prod_{j=1}^{n} m_j \right) \cdot (n - 1) + \sum_{j=1}^{n} \sum_{i_1=1}^{m_1} \cdots \sum_{i_n=1}^{m_n} a_{ij_j}$$

37
3. Satisfiability

\[
\begin{align*}
&= (\prod_{j=1}^{n} m_j) \cdot (n - 1) + \sum_{j=1}^{n} \sum_{i=1}^{m_j} a_{ji} \cdot \prod_{k=1}^{n} m_k \\
&= (\prod_{j=1}^{n} m_j) \cdot (n - 1) + \sum_{j=1}^{n} \sum_{i=1}^{m_j} \frac{a_{ji}}{m_j} \cdot \prod_{k=1}^{n} m_k \\
&= (\prod_{j=1}^{n} m_j) \cdot (n - 1) + (\prod_{k=1}^{n} m_k) \cdot \sum_{j=1}^{n} \sum_{i=1}^{m_j} a_{ji} \\
&= \left(\prod_{j=1}^{n} m_j\right) \cdot (n - 1) + \left(\prod_{k=1}^{n} m_k\right) \cdot \sum_{j=1}^{n} \frac{a_{ji}}{m_j} \\
&= \left(\prod_{j=1}^{n} m_j\right) \cdot (n - 1) + \left(\prod_{j=1}^{n} m_j\right) \cdot \sum_{i=1}^{m_j} \frac{a_{ji}}{m_j} \\
&= \left(\prod_{j=1}^{n} m_j\right) \cdot \left(\sum_{j=1}^{n} \frac{a_{ji}}{m_j} + (n - 1)\right) \\
&= \left(\prod_{j=1}^{n} (o_j + 1)\right) \cdot \left(\sum_{j=1}^{n} \frac{a_{ji}}{o_j + 1} + (n - 1)\right)
\end{align*}
\]

Explanation:
(1) by definition, as explained above
(2) \((n - 1)\) is added \(\prod_{j=1}^{n} m_j\) times
(3) sum reordering
(4) \(a_{ji}\) is added exactly \(\prod_{k=1}^{n} m_k\) times
(5) \(\prod_{k=1}^{n} m_k = \frac{1}{\prod_{j=1}^{n} m_j} \prod_{k=1}^{n} m_k\)
(6) distributivity
(7) distributivity
(8) \(a_j = \sum_{i=1}^{m_j} a_{ji}\), see above
(9) renaming
(10) distributivity
(11) \(o_j = m_j - 1\), i.e., \(m_j = o_j + 1\), see above

Altogether, the total number of disjunctions and conjunctions of the current expression can be calculated as follows:

\[
(o, a) = \left(\prod_{j=1}^{n} (o_j + 1)\right) - 1, \quad \left(\prod_{j=1}^{n} (o_j + 1)\right) \cdot \left(\sum_{j=1}^{n} \frac{a_{ji}}{o_j + 1} + (n - 1)\right)
\]
Once again, we showed that we actually only need to know the number of disjunctions and conjunctions of each sub-expression to accurately calculate the total number of disjunctions and conjunctions of the current expression.

Example 7 (Example for dnfLength (see algorithm \[1\]) Consider the following expression:

\[(\hat{b}_1 \lor (\hat{b}_2 \land \hat{b}_3)) \land \hat{b}_4) \lor (\hat{b}_5 \land \hat{b}_6 \lor \hat{b}_7 \lor \hat{b}_8) \land \hat{b}_9\]

Its straightforward disjunctive normal form contains 4 disjunctions and 9 conjunctions:

\[
\begin{aligned}
&\equiv ((\hat{b}_1 \land \hat{b}_4) \lor (\hat{b}_2 \land \hat{b}_3 \land \hat{b}_4)) \lor ((\hat{b}_5 \land \hat{b}_6 \land \hat{b}_9) \lor (\hat{b}_5 \land \hat{b}_7 \land \hat{b}_9) \lor (\hat{b}_5 \land \hat{b}_8 \land \hat{b}_9)) \\
&\equiv (\hat{b}_1 \land \hat{b}_4) \lor (\hat{b}_2 \land \hat{b}_3 \land \hat{b}_4) \lor (\hat{b}_5 \land \hat{b}_6 \land \hat{b}_9) \lor (\hat{b}_5 \land \hat{b}_7 \land \hat{b}_9) \lor (\hat{b}_5 \land \hat{b}_8 \land \hat{b}_9) \\
&\equiv (4, 9)
\end{aligned}
\]

If we forgo the transformation into the disjunctive normal form and apply the algorithm dnfLength (see algorithm \[1\]), we obtain the same result:

The algorithm dnfLength (see algorithm \[1\]) has linear runtime with respect to the length of the considered expression \(\hat{\beta}\), i.e., \(\text{dnfLength} \in \mathcal{O}(|\hat{\beta}|)\). The algorithm dnfLength can be used as a reliable heuristic to select the fastest solver for an expression. For a given expression, first of all, we remove all negations. Then, we calculate the total number of disjunctions and the total number of conjunctions of the straightforward disjunctive normal form. The measure of the heuristic is the logarithm of the sum of those two numbers, i.e., \(\ln(1 + o + \alpha)\).
3. Satisfiability

Experiments have shown that for values smaller than 6, the DNF solver should be chosen. For values between 6 and 23, the Yices solver [DM06] is best. And for values greater than 23, MathSAT 5 [CGSS13] is the fastest.

We have used the actual expressions of the prototype IQCache, a top-k semantic cache, during the Yahoo! Cloud Serving Benchmark [CST+10] and the Twitter benchmark [CHBG10] to evaluate the hybrid solver. Using the described heuristic, the hybrid solver is able to distinctly outperform all other considered solvers (see figure 3.1).

![Figure 3.1.](image)

Figure 3.1.: Performance of the hybrid solver in comparison with Yices, MathSAT 5 and Z3 for expressions of the prototype IQCache during Yahoo! Cloud Serving Benchmark and Twitter benchmark
3.3. Three-valued Logic and SQL

Because of the incorporation of *null values*, an expression in SQL can have one of the following three logical results: true, false and unknown. Therefore, expressions in SQL must use three-valued logic.

### 3.3.1. Conjunction, Disjunction, Negation

In fact, the logic of SQL is based on the common subset of Kleene logic and Łukasiewicz logic [Kle50, Luk70, Kle03]. The definitions of conjunction, disjunction and negation in three-valued logic are shown in table 3.6, table 3.7 and table 3.8.

<table>
<thead>
<tr>
<th>$\hat{a} \land \hat{b}$</th>
<th>false</th>
<th>unknown</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>false</td>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 3.6.: Conjunction in three-valued logic (i.e., $\land$)

<table>
<thead>
<tr>
<th>$\hat{a} \lor \hat{b}$</th>
<th>false</th>
<th>unknown</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 3.7.: Disjunction in three-valued logic (i.e., $\lor$)

<table>
<thead>
<tr>
<th>$\hat{a}$</th>
<th>$\neg \hat{a}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3.8.: Negation in three-valued logic (i.e., $\neg$)

### 3.3.2. Isunknown

In addition, we define the unary operator *isunknown*, written $u$, which checks whether an expression is unknown. This operator makes the three-valued logic functionally
3. Satisfiability

complete [Dat95]. It is defined in extension F571 of SQL92 [MS93]. It is implemented in PostgreSQL since version 7.2 [Pos02]. The definition of `isunknown` is shown in table 3.9.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td><code>false</code></td>
</tr>
<tr>
<td>unknown</td>
<td><code>true</code></td>
</tr>
<tr>
<td>true</td>
<td><code>false</code></td>
</tr>
</tbody>
</table>

Table 3.9.: `isunknown` in three-valued logic (i.e., `u`)

The operator `isunknown` has the interesting effect that it absorbs negations.

**Proposition 3 (Unknown and negation)** If we apply the operator `isunknown` to a negated expression `¬ˆa`, we can omit the negation and apply `isunknown` directly to `ˆa`:

\[ u(¬ˆa) \equiv uˆa \]

**Proof.** This proposition can be verified by truth table 3.10.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>¬ˆa</th>
<th>u(¬ˆa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>unknown</td>
<td>true</td>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3.10.: Truth table that proves \( u(¬ˆa) \equiv uˆa \)

□

Similar to the negation that can be pushed to the leaves of the expression using de Morgan’s law [SB00], an equivalence can be used to push the operator `isunknown` to the leaves of the expression.

First, we consider the combination of `isunknown` and a conjunction:

**Proposition 4 (Isunknown and conjunction)**

\[ u(ˆa ∧ ˆb) = (uˆa ∧ ˆb) ∨ (uˆa ∧ u ˆb) \lor (uˆa ∧ u ˆb) \]

**Proof.** The correctness of this proposition is demonstrated by using a truth table, see table 3.11.

□

Secondly, we look at the combination of `isunknown` and disjunction:

**Proposition 5 (Isunknown and disjunction)**

\[ u(ˆa ∨ ˆb) = (uˆa ∨ ˆb) ∨ (uˆa ∧ ¬ˆb) \lor (uˆa ∧ u ˆb) \]

**Proof.**
### Table 3.11.: Truth table that proves $u(\hat{a} \land \hat{b}) \equiv (u\hat{a} \land \hat{b}) \lor (\hat{a} \land u\hat{b}) \lor (u\hat{a} \land u\hat{b})$
3. Satisfiability

\[ u(\hat{a} \lor \hat{b}) \equiv u(\neg \hat{a} \land \neg \hat{b}) \]
\[ \equiv u(\neg \hat{a} \land \neg \hat{b}) \]
\[ \equiv (u(\neg \hat{a}) \land \neg \hat{b}) \lor (\neg \hat{a} \land u(\neg \hat{b})) \lor (u(\neg \hat{a}) \land u(\neg \hat{b})) \]
\[ \equiv (u\hat{a} \land \neg \hat{b}) \lor (\neg \hat{a} \land u\hat{b}) \lor (u\hat{a} \land u\hat{b}) \]

**Explanation:**
(1) de Morgan
(2) \( u(\neg \hat{a}) \equiv u\hat{a} \) (see proposition 3)
(3) \( u(\hat{a} \land \hat{b}) = (u\hat{a} \land \hat{b}) \lor (u\hat{a} \land u\hat{b}) \lor (u\hat{a} \land u\hat{b}) \) (see proposition 4)
(4) \( u(\neg \hat{a}) \equiv u\hat{a} \) (see proposition 3)

In addition, please note the following behavior of the composition of *isunknown* with itself.

**Proposition 6 (Composition of *isunknown* with itself)** The composition of *isunknown* with itself is a contradiction:

\[ u(u(\hat{a})) \equiv false \]

**Proof.** The inequality \( u(\hat{a}) \neq unknown \) always holds by definition of the operator \( u \), see table 3.9. Hence, the proposition follows.

3.3.3. Partial Equivalence

**Example 8 (Partial equivalence)** Consider the following two SQL queries \( p_1 \) and \( p_2 \) that both return all posts that have been changed today:

```
1 SELECT p.*
2 FROM post p
3 WHERE p.changed >= TODAY
```

Listing 3.1: Query \( p_1 \)

```
1 SELECT p.*
2 FROM post p
3 WHERE (p.changed >= TODAY)
4 AND
5 ((p.changed >= TODAY) IS NOT UNKNOWN)
```

Listing 3.2: Query \( p_2 \)

The *where* conditions of both queries \( p_1 \) and \( p_2 \) can be straightforwardly translated into three-valued logic:

\[ \hat{p}_1 = changed \geq today \]
\[ \hat{p}_2 = (changed \geq today) \land \neg u(changed \geq today) \]
To analyze these expressions, we partition the value of the attribute *changed* into three classes. First of all, it can be today. Secondly, it can be somewhere in the past. And finally, if the post has never been changed, it may be set to *null*.

Therewith, we evaluate the expressions $\hat{p}_1$ and $\hat{p}_2$ for the different values of the attribute *changed* (see table 3.13). If the post has been changed today, both of them will evaluate to true. On the other hand, if it has been changed somewhere before today, they will identically evaluate to false. But if the value of the attribute *changed* is *null*, expression $\hat{p}_1$ will evaluate to unknown and expression $\hat{p}_2$ will evaluate to false.

<table>
<thead>
<tr>
<th>value of <em>changed</em></th>
<th>$\hat{p}_1$</th>
<th>$\hat{p}_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>not today</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>⊥ (null)</td>
<td>unknown</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3.13.: Expressions $\hat{p}_1$ and $\hat{p}_2$, an example for partial equivalence

Since expressions $\hat{p}_1$ and $\hat{p}_2$ evaluate to true for the same tuples, they describe the same set of tuples. And queries $p_1$ and $p_2$ will always return the same results. Therefore, queries $q_1$ and $q_2$ can be considered identical. In contrast, their *where* conditions $\hat{p}_1$ and $\hat{p}_2$ are not semantically equivalent, because they evaluate differently if the attribute *changed* is *null* – unknown or false, respectively.

To work with this kind of equality, we need a special type of equivalence that only demands that the considered expressions evaluate to the same result for a given truth value. We call it partial equivalence:

**Definition 5 (Partial equivalence)** Two expressions $\hat{a}$ and $\hat{b}$ are partially equivalent or $p$-equivalent (with $p \in \{\text{true}, \text{unknown}, \text{false}\}$), denoted by $\hat{a} \equiv_p \hat{b}$, iff.

$$\forall \mu : (\hat{a}[\mu] \equiv_p \hat{b}[\mu]) \iff p.$$  

**Example 9 (Partial equivalence – continued)** The expressions $\hat{p}_1$ and $\hat{p}_2$ are true-equivalent, i.e., $\hat{p}_1 \equiv_{\text{true}} \hat{p}_2$.

True-equivalence becomes important when we describe sets of tuples with three-valued logic. True-equivalent expressions describe the same sets of tuples. And sometimes, expressions can be simplified to a far greater extent using the weaker true-equivalence instead of semantic equivalence.

Finally, please note that partial equivalence plays no role in two-valued logic, because, here, partial equivalence and semantic equivalence are trivially identical.
3. Satisfiability

3.4. Satisfiability and Three-valued Logic

As discussed in section 3.2, SMT solvers determine the satisfiability of expressions over various domains. But they use two-valued logic. However, as discussed in section 3.3, expressions in SQL need three-valued logic. In this section, we outline how SMT solvers can be utilized to find overlaps and subsumptions of expressions in three-valued logic.

3.4.1. Overlaps

Consider two queries, e.g., query $q_1$ and query $q_2$ of the running example (see figure 3.2).

![Figure 3.2.: Overlap of two queries (e.g., query $q_1$ overlaps with query $q_2$)](image)

We want to find out if the where conditions of query $q_1$ and query $q_2$ overlap. Two expressions do overlap if there can exist at least one tuple that satisfies both of them.

**Definition 6 (Overlap)** Two expressions $\hat{a}$ and $\hat{b}$ in three-valued logic overlap iff.

$$\exists \mu : (\hat{a}[\mu] \equiv \text{true}) \land (\hat{b}[\mu] \equiv \text{true})$$

The function $f^{\text{true}}$ (also called NTT, i.e., null to true) can be used to convert a three-valued expression into a two-valued one [BG04, BG05]. If, for a given tuple, the initial expression evaluates to unknown, the resulting two-valued expression after applying the function $f^{\text{true}}$ will evaluate to false (see table 3.14).

<table>
<thead>
<tr>
<th>$\hat{q}[\mu]$</th>
<th>$f^{\text{true}}(\hat{q})[\mu]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>unknown</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 3.14.: The function $f^{\text{true}}$ converts expressions in three-valued logic to expressions in two-valued logic

With function $f^{\text{true}}$, an SMT solver can check if two expressions in three-valued logic overlap.
3.4. Satisfiability and Three-valued Logic

**Proposition 7 (Overlap and SMT solver)** Whether two expressions \( \hat{a} \) and \( \hat{b} \) that use three-valued logic overlap can be determined by an SMT solver that operates in two-valued logic by evaluating if the following expression is satisfiable:

\[
f^{\text{true}}(\hat{a} \land \hat{b})
\]

**Proof.**

\[
\exists \mu : \left( (\hat{a}[\mu] \equiv \text{true}) \land (\hat{b}[\mu] \equiv \text{true}) \right) \\
\Leftrightarrow \exists \mu : \left( (\hat{a}[\mu] \land \hat{b}[\mu]) \equiv \text{true} \right) \\
\Leftrightarrow \hat{a} \land \hat{b} \text{ is satisfiable in three-valued logic} \\
\Leftrightarrow f^{\text{true}}(\hat{a} \land \hat{b}) \text{ is satisfiable in two-valued logic}
\]

\[\square\]

**Explanation:**

(1) definition of conjunction (see table 3.6)
(2) definition of satisfiability
(3) definition of function \( f^{\text{true}} \) (see table 3.14)

3.4.2. Subsumptions

Consider two queries, e.g., query \( q_2 \) and query \( q_3 \) of the running example (see figure 3.3).

\[
\begin{aligned}
q_2 \\
\cap \\
q_3
\end{aligned}
\]

Figure 3.3.: Subsumption of two queries (e.g., query \( q_2 \) subsumes query \( q_3 \))

We want to find out if the *where* condition of query \( q_2 \) subsumes the *where* condition of query \( q_3 \). An expression is subsumed by another expression if all tuples that satisfy the first expression also satisfy the second one.
3. Satisfiability

Definition 7 (Subsumption) An expression $\hat{\alpha}$ in three-valued logic subsumes an expression $\hat{\beta}$ in three-valued logic iff.

$$\forall \mu: \left( (\hat{b}[\mu] \equiv \text{true}) \rightarrow (\hat{a}[\mu] \equiv \text{true}) \right).$$

Again, we can take advantage of the function $f_{\text{true}}$ (see table 3.14).

Proposition 8 (Subsumption and SMT solver) Whether an expression $\hat{\alpha}$ in three-valued logic subsumes an expression $\hat{\beta}$ in three-valued logic can be determined by an SMT solver that operates in two-valued logic by evaluating if the following expression is not satisfiable:

$$f_{\text{true}}(\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha}))$$

Proof.

$$\forall \mu: ((\hat{b}[\mu] \equiv \text{true}) \rightarrow (\hat{a}[\mu] \equiv \text{true}))$$

(1) $$\neg \exists \mu: (\neg(\hat{b}[\mu] \equiv \text{true}) \rightarrow (\hat{a}[\mu] \equiv \text{true}))$$

(2) $$\neg \exists \mu: (\neg(\hat{b}[\mu] \equiv \text{true}) \lor (\hat{a}[\mu] \equiv \text{true}))$$

(3) $$\neg \exists \mu: ((\hat{b}[\mu] \equiv \text{true}) \land \neg(\hat{a}[\mu] \equiv \text{true}))$$

(4) $$\neg \exists \mu: (\hat{b}[\mu] \equiv \text{true}) \land (\hat{a}[\mu] \equiv \text{unknown} \lor \neg \hat{a}[\mu] \equiv \text{false})$$

(5) $$\neg \exists \mu: ((\hat{b}[\mu] \equiv \text{true}) \land (\hat{a}[\mu] \equiv \text{unknown} \lor \neg \hat{a}[\mu] \equiv \text{true}))$$

(6) $$\neg \exists \mu: ((\hat{b}[\mu] \equiv \text{true}) \land (u(\hat{a})[\mu] \equiv \text{true} \lor \neg \hat{a}[\mu] \equiv \text{true}))$$

(7) $$\neg \exists \mu: ((\hat{b}[\mu] \equiv \text{true}) \land ((u(\hat{\alpha}) \lor \neg \hat{\alpha})[\mu] \equiv \text{true}))$$

(8) $$\neg \exists \mu: (\neg(\exists \mu: (\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha}))[\mu] \equiv \text{true}))$$

(9) $$\neg(\exists\mu: ((\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha})))[\mu] \equiv \text{true}))$$

(10) $$\neg(\exists\mu: ((\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha})))[\mu] \equiv \text{true}))$$

(11) $$\neg(\exists\mu: ((\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha})))[\mu] \equiv \text{true}))$$

(12) $$\neg(\exists\mu: ((\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha})))[\mu] \equiv \text{true}))$$

$$f_{\text{true}}(\hat{b} \land (u(\hat{\alpha}) \lor \neg \hat{\alpha}))$$ is not satisfiable in two-valued logic

Explanation:

(1) $\forall \mu (g(\mu)) \equiv \neg \exists \mu: \neg(g(\mu))$ for any $g$

(2) definition of $\rightarrow$ in two-valued logic

(3) de Morgan

(4) in three-valued logic, if an expression is not evaluated to true, it must be evaluated to false or unknown

(5) definition of negation in three-valued logic (see table 3.8)
3.4. Satisfiability and Three-valued Logic

(6) definition of isunknown in three-valued logic (see table 3.9)
(7) definition of disjunction in three-valued logic (see table 3.7)
(8) definition of conjunction in three-valued logic (see table 3.6)
(9) addition of parentheses of negation
(10) definition of satisfiability
(11) definition of function \( f \) (see table 3.14)
(12) definition of negation in two-valued logic
Part II.

Top-k Semantic Cache Design
4. Segments

A top-k semantic cache is a logical advancement of a semantic cache. In contrast to a semantic cache, a top-k semantic cache naturally supports top-k queries. This has effects on segments, operations on segments, and on query answering. In this chapter, we define and describe segments of a top-k semantic cache. In addition, set operations on segments are introduced.

4.1. Segment Definition

Similar to a semantic cache, a top-k semantic cache consists of disjoint segments that are originally based on answers to queries. In case of a top-k semantic cache, however, these are top-k queries. We will consider only projection-selection-join top-k queries, i.e., sub-queries and aggregations are not allowed.

4.1.1. Definition

We define a query $q$ as follows:

**Definition 8 (Query)** The tuple

\[ q = (A_{select}, A_{from}, A_{where}, A_{orderby}, R, \hat{q}, O, k) \]

describes a sorted and limited select-project-join query.

The attribute sets $A_{select}$, $A_{from}$, $A_{where}$, and $A_{orderby}$ contain the attributes that are used in the corresponding query parts. $R$ is the set of all tables that were used in the $from$ clause. $\hat{q}$ is the $where$ condition. $O$ contains a description of the $order by$ clause. $k$ is the limit of the query.

**Example 10 (Description of query $q_1$)** Consider query $q_1$ (see listing 4.1), the first query of the running example (see section 1.2):

```
SELECT p.id, p.likes, p.created, p.title, p.author
FROM post p
WHERE (p.created >= LAST_WEEK) AND (p.likes >= 100)
ORDER BY p.likes DESC
```
4. Segments

Listing 4.1: Query \( q_1 \): a top-k query returning last week’s most liked posts

Query \( q_1 \) can be described by tuple \( q_1 = (A_{select}^1, A_{from}^1, A_{where}^1, A_{orderby}^1, R_1, \hat{q}_1, O_1, k_1) \) with the values as defined in table 4.1.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{select}^1 )</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>( A_{from}^1 )</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>( A_{where}^1 )</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>( A_{orderby}^1 )</td>
<td>{likes}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( {post} )</td>
</tr>
<tr>
<td>( \hat{q}_1 )</td>
<td>( (created \geq \text{last week}) \land (likes \geq 100) )</td>
</tr>
<tr>
<td>( O_1 )</td>
<td>((\text{likes}, \downarrow)) ) (for notation, see definitions 12 and 11)</td>
</tr>
<tr>
<td>( k_1 )</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.1.: Values of query \( q_1 \)

Based on the query definition (see definition 8), we can define a segment of a top-k semantic cache:

**Definition 9 (Segment of a top-k semantic cache)** The tuple

\[
\hat{s} = (A_{select}^s, A_{from}^s, A_{where}^s, A_{orderby}^s, A_{initial}^s, R, \hat{s}, O, k, \hat{i}, \lambda, c, T)
\]

describes a segment of a top-k semantic cache.

The definition of a segment expands on the definition of a query. The attribute set \( A_{select}^s \), \( A_{from}^s \), \( A_{where}^s \), and \( A_{orderby}^s \) still contain the attributes that are used in the corresponding query parts. But the attribute sets must now always satisfy the following conditions:

1. \( A_{select}^s \supseteq \text{primary}(A_{from}^s) \)
2. \( A_{select}^s \supseteq A_{orderby}^s \)

The first condition states that all primary attributes of all tables that are used in the \textit{from} clause must be contained in the \textit{select} clause. This condition is needed to allow the cache to load additional non-primary columns at a later point (see section 6.2.2). The second condition is required to be able to calculate a condition that describes the tuples that have been loaded for the segment (see proposition 10).
4.1. Segment Definition

$R$ is the set of base tables of the segment. $\hat{s}$ is the where condition. $O$ contains a description of the order by clause. For a segment, the ordering that is defined by the order by clause must be a total order. This is no limitation and can be easily achieved by adding all primary attributes of all referenced tables (i.e., $\text{primary}(A_{\text{from}})$) at the end of the order by of a query. $k$ contains the number of tuples in the segment.

The variables $\hat{i}$, $A_{\text{initial}}$, $\lambda$, $c$, and $T$ are cache variables that are needed for cache management and that do not have a corresponding part in a top-k query.

When a new segment is created, tuples might be moved from other segments to the newly created one. This is necessary, because all segments must be disjoint. One way to ensure this property is to move all tuples from overlaps of other segments with the new one into the new one. The condition $\hat{i}$ describes the tuples that have been moved into the segment when it was initially created. The attribute set $A_{\text{initial}}$ contains all attributes that are used in $\hat{i}$.

$\lambda$ is the tuple that has been loaded last from the server. Please note that tuples that are moved into a segment from other ones will not change the $\lambda$ of the segment.

c is a flag that indicates whether the segment has been loaded completely. This flag will be used for optimization. $T$ is the set of tuples that have been loaded.

An overview of the parts of a segment of a top-k semantic cache can be found in table 4.2.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{\text{select}}$</td>
<td>Attributes of select clause</td>
</tr>
<tr>
<td>$A_{\text{from}}$</td>
<td>Attributes of all tables of from clause</td>
</tr>
<tr>
<td>$A_{\text{where}}$</td>
<td>Attributes that were used in the where condition</td>
</tr>
<tr>
<td>$A_{\text{order by}}$</td>
<td>Attributes that were used in the order by clause</td>
</tr>
<tr>
<td>$A_{\text{initial}}$</td>
<td>Attributes that are used in the condition $\hat{i}$ that describes initially loaded tuples</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>Tables that were used in the from clause</td>
</tr>
<tr>
<td>$\hat{s}$</td>
<td>where condition</td>
</tr>
<tr>
<td>$O$</td>
<td>Description of order by clause</td>
</tr>
<tr>
<td>$k$</td>
<td>Number of elements in the segment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{i}$</td>
<td>Condition that describes initially loaded tuples</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Tuple that has been loaded last</td>
</tr>
<tr>
<td>$c$</td>
<td>Has the segment been loaded completely?</td>
</tr>
<tr>
<td>$T$</td>
<td>Tuples (that have been loaded)</td>
</tr>
</tbody>
</table>

Table 4.2.: Overview of parts of a top-k semantic cache segment
4. Segments

Example 11 (Segment $s_1$ for query $q_1$) We use the sample instance from the running example (see table 1.1 in section 1.2). And we assume an empty cache. If we execute query $q_1$ using the sample instance, we get the query result in table 4.3.

<table>
<thead>
<tr>
<th>T1</th>
<th>id</th>
<th>likes</th>
<th>created</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\mu_3$</td>
<td>3</td>
<td>250</td>
</tr>
<tr>
<td>2</td>
<td>$\mu_4$</td>
<td>4</td>
<td>240</td>
</tr>
<tr>
<td>3</td>
<td>$\mu_{10}$</td>
<td>10</td>
<td>230</td>
</tr>
<tr>
<td>4</td>
<td>$\mu_7$</td>
<td>7</td>
<td>220</td>
</tr>
<tr>
<td>5</td>
<td>$\mu_1$</td>
<td>1</td>
<td>210</td>
</tr>
<tr>
<td>6</td>
<td>$\mu_{19}$</td>
<td>19</td>
<td>200</td>
</tr>
<tr>
<td>7</td>
<td>$\mu_{14}$</td>
<td>14</td>
<td>190</td>
</tr>
<tr>
<td>8</td>
<td>$\mu_8$</td>
<td>8</td>
<td>180</td>
</tr>
<tr>
<td>9</td>
<td>$\mu_9$</td>
<td>9</td>
<td>170</td>
</tr>
<tr>
<td>10</td>
<td>$\mu_2$</td>
<td>2</td>
<td>160</td>
</tr>
</tbody>
</table>

Table 4.3.: Query result of query $q_1$ using sample instance of running example

Based on this query result, we can create a top-k semantic cache segment $s_1$ for query $q_1$ (see figure 4.1). Most common variables of query $q_1$ and the corresponding segment $s_1$ have identical values (see example 10). According to definition 9, the order $O$ must be a total order. But an order based on the attribute likes is not a total order, because the attribute likes is not even a primary key or attribute with unique values of the relation post. Therefore, its values are also not unique.

But we can add the attribute id (i.e., $((\text{likes}, \downarrow), (\text{id}, \uparrow))$) to the order by clause to create a total order. The notation for the order by is defined in definitions 12 and 11.

Since we assumed an empty cache, no tuples from other segments need to be moved into segment $s_1$ upon creation. Therefore, $\hat{c}_1$ is false and $A_{\text{init}al}$ is the empty set.

The tuple $\lambda_1$, which is the tuple that has been loaded last, is set to $\mu_2$, because $\mu_2$ is the last tuple from the query result. Hence, it can be found in the last row of table 4.3. $T_1$ contains all tuples that have been loaded.

The query has not been completely loaded because, for example, tuple $\mu_6 = (6, 150, \text{last week}, \ldots)$ of the sample instance satisfies the where condition of query $q_1$, but it is not returned as part of the query result because it would be at position 11 of the top-10 query $q_1$. Therefore, $c_1$ is false.

Table 4.4 provides an overview of the variables of the segment and their discussed values.

4.1.2. Conditions

Based on the order by clause $O$ and the last loaded tuple $\lambda$, we can generate the following conditions:
4.1. Segment Definition

Figure 4.1.: Segment $s_1$ for query $q_1$

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1^{select}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_1^{from}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_1^{where}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_1^{orderby}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_1^{initial}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>{post}</td>
</tr>
<tr>
<td>$\hat{s}_1$</td>
<td>(created $\geq$ last week) $\land$ (likes $\geq$ 100)</td>
</tr>
<tr>
<td>$O_1$</td>
<td>((likes, ↓), (id, ↑))</td>
</tr>
<tr>
<td>$k_1$</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{c}_1$</td>
<td>false</td>
</tr>
<tr>
<td>$\lambda_1$</td>
<td>$\mu_0[\text{id, likes, created}] = (2, 160, \text{last week})$ (see table 4.3)</td>
</tr>
<tr>
<td>$c_1$</td>
<td>false</td>
</tr>
<tr>
<td>$T_1$</td>
<td>${\mu_1, \mu_2, \mu_3, \mu_4, \mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}}$ (see table 4.3)</td>
</tr>
</tbody>
</table>

Table 4.4.: Values of segment $s_1$
4. Segments

Definition 10 (Order by condition) The condition \( \hat{o} \leq \lambda \) describes the tuples that are smaller or equal to \( \lambda \) according to the order by \( O \). Respectively, the condition \( \hat{o} > \lambda \) characterizes the tuples that are greater than \( \lambda \) with respect to the order by \( O \).

Note that all conditions in this chapter use three-valued logic, because SQL is based on it (see section 3.3).

Proposition 9 (Tautology of order by conditions) The conditions \( \hat{o} \leq \lambda \) and \( \hat{o} > \lambda \) are antipodes and their disjunction contains all tuples, i.e., the statements \( \neg \hat{o} \leq \lambda \equiv \hat{o} > \lambda \) and \( \hat{o} \leq \lambda \lor \hat{o} > \lambda \equiv \text{true} \) apply.

Proof. Because the order by \( O \) is a total ordering by definition 9, for every tuple \( \mu \), it can be decided if \( \mu \leq \lambda \) or if \( \mu > \lambda \). Therefore, neither condition \( \hat{o} \leq \lambda \) nor condition \( \hat{o} > \lambda \) can be evaluated to unknown. \( \square \)

With these conditions, we can describe the tuples of a segment \( s \) (see figure 4.2).

Initially, a segment \( s \) contains the tuples described by the expression \( \hat{i} \land \hat{s} \). These tuples are not necessarily the smallest tuples according to the order \( O \). But they had to be moved into segment \( s \) to avoid overlaps of this segment and other segments. The expression \( \hat{i} \) allows us to prevent these tuples from being transferred to the client again. The tuples that are transferred from the server just continuously fill in the gaps between the tuples that have been initially moved into the segment.

We define the expression \( \hat{i} \) such that the restriction to segment \( s \) is implicit, i.e., that the expression does not need to contain the constraint \( \hat{s} \) (see figure 4.3). That way, if, for example, a segment \( s_1 \) initially overlaps with a segment \( s_2 \), the condition \( \hat{i} \) can be set to \( \hat{s}_2 \land \hat{s}_1 \) instead of \( \hat{s}_2 \land \hat{s}_1 \). Overall, this definition of \( \hat{i} \) leads to simpler expressions and, therefore, faster query processing.

The tuples of the gaps between the initially loaded tuples \( \hat{i} \land \hat{s} \) are loaded according to the order \( O \). For better understanding, imagine a bucket (\( \hat{s} \)) containing stones (\( \hat{i} \land \hat{s} \))
4.1. Segment Definition

Figure 4.3.: Conditions of a segment of a top-k semantic cache with implicitly defined \( \hat{i} \) that slowly fills with water. The water flows around the stones and occupies all the gaps between them. Then, the expression \( \hat{o} \leq \lambda \land \hat{s} \) describes the current water-level, i.e., the tuples of the segment than can be used to answer the corresponding top-k query.

Let \( l \) be the number of tuples in the completely filled area \( \hat{o} \leq \lambda \land \hat{s} \) of the segment, i.e., \( l = |\sigma_{\hat{o} \leq \lambda \land \hat{s}}(x_r \in R^r)| \). Then, the top-k query limit by \( \pi_A \) \( \sigma_{\hat{s}}(x_r \in R^r) \) can be answered correctly by segment \( s \).

Since \( \hat{i} \land \hat{s} \) describes the initially loaded tuples and \( \hat{o} \leq \lambda \land \hat{s} \) describes the area containing the tuples that have been transferred from the server, we can conclude the following propositions:

**Proposition 10 (Loaded tuples of segment)** The expression \((\hat{o} \leq \lambda \lor \hat{i}) \land \hat{s}\) describes the tuples that have been loaded for segment \( s \).

**Proposition 11 (Not loaded tuples of segment)** The expression \((\hat{o} > \lambda \land (\neg \hat{i} \lor \hat{u} \hat{i}) \land \hat{s}\) describes the tuples that would be contained in segment \( s \), but have not yet been loaded.

**Proof.**

\[
\begin{align*}
( (\hat{o} \leq \lambda \lor \hat{i}) \land \hat{s}) & \neq \text{true} \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) \lor (\hat{i} \land \hat{s}) ) & \neq \text{true} \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land ( (\hat{i} \land \hat{s}) ) \neq \text{true} ) \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land \hat{s} \land ( (\hat{i} \land \hat{s}) ) \neq \text{true} ) \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land \hat{s} \land ( (\hat{i} \land \hat{s}) ) \neq \text{true} \lor (u\hat{i} \land \hat{s}) ) \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land \hat{s} \land ( (\hat{i} \land \hat{s}) ) \neq \text{true} \lor (u\hat{i} \land \hat{s}) \lor (\hat{i} \land u\hat{s}) ) \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land \hat{s} \land ( (\hat{i} \land \hat{s}) ) \lor (\hat{i} \land u\hat{s}) \lor (u\hat{i} \land u\hat{s}) ) \land \hat{s} \\
\equiv ( (\hat{o} \leq \lambda \land \hat{s}) ) & \neq \text{true} \land \hat{s} \land ( (\hat{i} \land \hat{s}) ) \lor (\hat{i} \land u\hat{s}) \lor (u\hat{i} \land u\hat{s}) ) \land \hat{s} \\
\end{align*}
\]
4. Segments

\begin{align*}
&\equiv ( (\widehat{\alpha} \leq \lambda \land \widehat{s} \neq \text{false}) \land \widehat{s} ) \land \left( (\neg \widehat{i} \lor u\widehat{i}) \land \widehat{s} \right) \\
&\equiv ( (\neg (\widehat{\alpha} \leq \lambda \land \widehat{s}) \lor u(\widehat{\alpha} \leq \lambda \land \widehat{s})) \land \widehat{s} ) \land \left( (\neg \widehat{i} \lor u\widehat{i}) \land \widehat{s} \right) \\
&\equiv \text{true} \quad ( (\neg \widehat{\alpha} \leq \lambda \lor \neg \widehat{s} \lor (u\widehat{\alpha} \leq \lambda \land \widehat{s}) \lor (u\widehat{\alpha} \leq \lambda \land u\widehat{s}) \lor (u\widehat{\alpha} \leq \lambda \land u\widehat{s}) ) ) \land \left( (\neg \widehat{i} \lor u\widehat{i}) \land \widehat{s} \right) \\
&\equiv \text{true} \quad ( (\neg \widehat{\alpha} \leq \lambda \land \widehat{s}) \land \left( (\neg \widehat{i} \lor u\widehat{i}) \land \widehat{s} \right) \\
&\equiv (\neg \widehat{\alpha} \leq \lambda \land (\neg \widehat{i} \lor u\widehat{i})) \land \widehat{s} \\
\end{align*}

Explanation:
(1) distributivity
(2) \((\widehat{x} \lor \widehat{y}) \neq \text{false}) \equiv ( (\widehat{x} \neq \text{false}) \land (\widehat{y} \neq \text{false}) )\) for all \(\widehat{x}, \widehat{y}\)
(3) \(\widehat{x} \equiv \widehat{x} \land \widehat{x}\) for all \(\widehat{x}\), commutativity, and associativity
(4) \(\widehat{x} \neq \text{false}) \equiv (\neg \widehat{x} \lor u\widehat{x})\) for all \(\widehat{x}\)
(5) de Morgan, and \(u(\widehat{x} \land \widehat{y}) \equiv ((u\widehat{x} \land \widehat{y}) \lor (\neg \widehat{x} \land \neg \widehat{y}) \lor (u\widehat{x} \land \neg \widehat{y}) )\) for all \(\widehat{x}, \widehat{y}\) (see proposition 4)
(6) \(\neg \widehat{s} \land \widehat{s}, \widehat{s} \land u\widehat{s} \land \widehat{s}, \text{ and } u\widehat{s} \land u\widehat{s} \land \widehat{s}\) cannot be evaluated to true
(7) distributivity, associativity, and \(\widehat{x} \land \widehat{x} \equiv \widehat{x}\) for all \(\widehat{x}\)
(8) see (4)
(9) see (5)
(10) \(u\widehat{\alpha} \leq \lambda \equiv \text{false}\) (see proposition [4]), hence, \(u\widehat{\alpha} \leq \lambda \land \widehat{s} \equiv \text{false}\) as well as \(u\widehat{\alpha} \leq \lambda \land u\widehat{s} \equiv \text{false}\), and \(\neg \widehat{s} \land \widehat{s}\) as well as \(\widehat{\alpha} \leq \lambda \land u\widehat{s} \land \widehat{s}\) cannot be evaluated to true
(11) \(\neg \widehat{\alpha} \leq \lambda \equiv \neg \widehat{\alpha} \leq \lambda\) (see proposition [6])
(12) distributivity

**Corollary 12 (Not loaded tuples of segments)** If all attributes of \(\widehat{i}\) are defined as *not null*, the expression \(u\widehat{i}\) always evaluates to false. Therefore, in that case, the expression \((\neg \widehat{\alpha} \leq \lambda \land \neg \widehat{i}) \land \widehat{s}\) describes the tuples that would be contained in segment \(\widehat{s}\), but have not yet been loaded.

To describe the tuples that have been loaded for a segment \(\widehat{s}\), we obviously need to be able to generate the condition \(\widehat{\alpha} \leq \lambda\).

Please note that we have not made any restrictions for the *order by* clause. And sure enough, as long as you define how the condition \(\widehat{\alpha} \leq \lambda\) can be created based on the ordering \(O\) and the last loaded tuple \(\lambda\), the top-k semantic cache can process the query.

4.1.3. Order By Condition of Lexicographical Orderings

We will have a closer look at how the condition \(\widehat{\alpha} \leq \lambda\) can be created for *order by* clauses that consists of attribute-based lexicographical orderings, which can be ascending and descending.
4.1. Segment Definition

Definition 11 (Order by component) An order by component $o$ of a query or segment with attribute set $A_{\text{from}}$ is a pair of the set $A_{\text{from}} \times \{\uparrow, \downarrow\}$. The arrow $\uparrow$ denotes an ascending order. Respectively, the arrow $\downarrow$ denotes a descending order.

Definition 12 (Order by) An order by $O$ of a given query or segment with attribute set $A_{\text{from}}$ is a sequence of order by components, i.e., $O = (o_1, \ldots, o_n) \in (A_{\text{from}} \times \{\uparrow, \downarrow\})^n$ for an $n \in \mathbb{N}$. Therefore, an order by describes a lexicographical order based on the used attributes.

Example 12 (Order by of segment $s_1$) The order by condition

\[
\text{ORDER BY likes DESC, id ASC}
\]

is represented by $((\text{likes}, \downarrow), (\text{id}, \uparrow))$.

Since SQL:2003 SQL supports stating the sort order of null values. Extension T611, which is called Elementary OLAP operations, describes that null values can be sorted before or after all other data by stating nulls first or nulls last [ISO03]. However, not all database systems implement this functionality. In this work, we will assume that null values are considered greater than all other values. I.e., in every ascending order, they are sorted after and, in every descending order, they are sorted before all other data, respectively.

Please note that our approach and algorithm 2 can be adapted to support nulls first and nulls last in a straightforward way, but would make the presentation of the algorithms needlessly longish.

The condition $\hat{o} \leq \lambda$ for an order by $O$ and a last loaded tuple $\lambda$ can be generated by algorithm 2.

For an empty order by clause, the algorithm returns true, because no ordering will only be allowed for segments that have been completely loaded.

If $O$ contains only one component $(a_1, d_1)$, and the corresponding value $\lambda[a_1]$ of the tuple $\lambda$ is not $\perp$ (i.e., null), the correct comparison will be created. That is $a_1 \leq \lambda[a_1]$ for an ascending order and $a_1 \geq \lambda[a_1]$ for a descending one, respectively. But if $\lambda[a_1]$ is $\perp$, true will be returned for an ascending order, because all values are considered less than or equal to null values. And $a_1$ is $\perp$ will be returned for a descending order, because only null values are greater or equal to null values.

The condition $\hat{o} \leq \lambda$ can be defined recursively if $O$ contains more than two components, because the order is lexicographic, i.e., components that are closer to the start of $O$ always have prevalence. Hence, if two tuples are ordered by the ordering defined by the first component of the order by, the other components need not to be considered. Only, if the tuples are not ordered by the ordering defined by the first component, the other components will be considered. Let therefore $O_2 = ((a_2, d_2), \ldots, (a_n, d_n))$ be the order by clause beginning from the second component.
4. Segments

Algorithm 2 $\hat{o}^{\leq \lambda}$: conditionOrderBy$(O, \lambda)$

Input: $O, \lambda$

Output: $\hat{o}^{\leq \lambda}$

1: $O = ((a_1, d_1), \ldots, (a_n, d_n))$ with $a_1, \ldots, a_n \in A^{from}$, $d_1, \ldots, d_n \in \{\uparrow, \downarrow\}$, and an $n \in \mathbb{N}$.
2: if $n = 0$ then
3: return true
4: else if $n = 1$ then
5: if $\lambda[a_1]$ is $\bot$ then
6: if $d_1 = \uparrow$ then
7: return true
8: else
9: return $a_1$ is $\bot$
10: end if
11: else
12: if $d_1 = \uparrow$ then
13: return $a_1 \leq \lambda[a_1]$
14: else
15: return $a_1 \geq \lambda[a_1]$
16: end if
17: end if
18: else
19: $O_2 = ((a_2, d_2), \ldots, (a_n, d_n))$
20: if $\lambda[a_1]$ is $\bot$ then
21: if $d_1 = \uparrow$ then
22: return conditionOrderBy$(O_2, \lambda)$
23: else
24: return $(a_1$ is $\bot) \land$ conditionOrderBy$(O_2, \lambda)$
25: end if
26: else
27: if $d_1 = \uparrow$ then
28: return $(a_1 < \lambda[a_1]) \lor ((a_1 = \lambda[a_1]) \land$ conditionOrderBy$(O_2, \lambda)$
29: else
30: return $(a_1$ is $\bot) \lor (a_1 > \lambda[a_1]) \lor ((a_1 = \lambda[a_1]) \land$ conditionOrderBy$(O_2, \lambda)$
31: end if
32: end if
33: end if
If \( \lambda[a_1] \) is \( \perp \) and the first component defines an ascending order, we need not create a condition for the first component. Because null values are considered greater than all other values, all values are less than or equal to null values. Therefore, the algorithm can continue with the recursive call \( \text{conditionOrderBy}(O_2, \lambda) \).

Accordingly, if \( \lambda[a_1] \) is \( \perp \), but the first component defines a descending order, \( a_1 \) must be \( \perp \), and the algorithm returns \( (a_1 \text{ is } \perp) \land \text{conditionOrderBy}(O_2, \lambda) \).

If \( \lambda[a_1] \) is not \( \perp \) and the first component defines an ascending order, the condition \( \hat{o} \leq \lambda \) can be calculated recursively by \( (a_1 \text{ is } \perp) \lor ((a_1 = \lambda[a_1]) \land \text{conditionOrderBy}(O_2, \lambda)) \).

A check \( (a_1 \text{ is } \not\perp) \) is not needed, because for a tuple \( \mu \) with \( \mu[a_1] \text{ is } \perp \), the expression will evaluate to unknown. Hence, it will not be part of the result set described by the expression.

Analogously, if \( \lambda[a_1] \) is not \( \perp \) and the first component defines a descending order, the condition \( \hat{o} > \lambda \) can be calculated recursively by \( (a_1 \text{ is } \perp) \lor (a_1 > \lambda[a_1]) \lor ((a_1 = \lambda[a_1]) \land \text{conditionOrderBy}(O_2, \lambda)) \).

Both the runtime of algorithm 2 and the size of the result are linear with respect to the length of the order by, i.e., in \( O(|O|) \).

The condition \( \hat{o} > \lambda \) can be calculated in two ways. First, since, according to proposition 9, the equivalence \( \hat{o} > \lambda \equiv \neg \hat{o} \leq \lambda \) holds, the condition \( \hat{o} > \lambda \) can be calculated by negating the result of algorithm 2, i.e., \( \neg \text{conditionOrderBy}(O, \lambda) \). Secondly, an algorithm similar to algorithm 2 can be designed that calculates the expression \( \hat{o} > \lambda \) directly. The implementation of the top-k semantic cache, IQCache (see chapter 7), uses the second approach.

### 4.1.4. Orderings Using Arbitrary Score Functions

In the previous section, we studied the construction of an order by condition for lexicographical orderings using algorithm conditionOrderBy(\( O, \lambda \)) (see algorithm 2). But we have also stated above that our approach did not make any restrictions for the order by clause. In this section, we will outline how the condition \( \hat{o} \leq \lambda \) can be created based on the ordering \( O \) and the last loaded tuple \( \lambda \) for orderings using arbitrary score functions.

---

**Example 13 (Order by condition \( \hat{o} \leq \lambda \) of segment s_1)** For a given order by clause \( ((\text{likes}, \downarrow), (\text{id}, \uparrow)) \) and tuple \( \lambda_1[\text{id,likes}] = (2, 160) \), algorithm 2 calculates:

\[
\hat{o} \leq \lambda_1 = \text{conditionOrderBy}(O_1, \lambda_1) = \text{conditionOrderBy}(((\text{likes}, \downarrow), (\text{id}, \uparrow)), \lambda_1) = (\text{likes is } \perp) \lor (\text{likes } > 160) \lor ((\text{likes } = 160) \land \text{conditionOrderBy}(((\text{id}, \uparrow)), \lambda_1)) = (\text{likes is } \perp) \lor (\text{likes } > 160) \lor ((\text{likes } = 160) \land (\text{id } \leq 2))
\]

The condition \( \hat{o} > \lambda \) can be calculated in two ways. First, since, according to proposition 9, the equivalence \( \hat{o} > \lambda \equiv \neg \hat{o} \leq \lambda \) holds, the condition \( \hat{o} > \lambda \) can be calculated by negating the result of algorithm 2, i.e., \( \neg \text{conditionOrderBy}(O, \lambda) \). Secondly, an algorithm similar to algorithm 2 can be designed that calculates the expression \( \hat{o} > \lambda \) directly. The implementation of the top-k semantic cache, IQCache (see chapter 7), uses the second approach.
4. Segments

Definition 13 (Attribute score function) Let \( a \) be an attribute of a relation. We call a function \( s : \text{dom}(a) \to \mathbb{R} \) a score function of the attribute \( a \).

Example 14 (Attribute score function) Consider the relation \( \text{post} \). The domain of attribute \( \text{likes} \) (and attribute \( \text{dislikes} \)) is the natural numbers \( \mathbb{N} \) as there can never be a negative number of likes or dislikes. Hence, the following function \( s : \mathbb{N} \to \mathbb{R} \) is a score function of the attribute \( \text{likes} \) (and attribute \( \text{dislikes} \)):

\[
s : \mathbb{N} \to \mathbb{R}, x \mapsto x^2
\]

Definition 14 (Combining function) A combining function \( c : \mathbb{R}^n \to \mathbb{R} \) combines the results of several score functions to a single result \( (n \in \mathbb{N}_1) \).

Definition 15 (Score function) Let \( T \) be a tuple space. We call a function \( f : T \to \mathbb{R} \) that maps every tuple \( \mu \in T \) to a real number \( f(\mu) \) a score function of the tuple space \( T \).

Suitable attribute score functions and a combining function can be used to calculate the score of a tuple.

Proposition 13 (Score function based on combining function) Let \( a_1, \ldots, a_n \) be attributes. Let \( T := \text{dom}(a_1) \times \text{dom}(a_n) \) be a tuple space. Let \( s_1 : \text{dom}(a_1) \to \mathbb{R}, \ldots, s_n : \text{dom}(a_n) \to \mathbb{R} \) be attribute score functions of the attributes \( a_1, \ldots, a_n \). Let \( c : \mathbb{R}^n \to \mathbb{R} \) be a combining function. Then, the following function \( f \) is a score function:

\[
f : T \to \mathbb{R}, \mu \mapsto c(s_1(\mu[a_1]), \ldots, s_n(\mu[a_n]))
\]

Example 15 (Score function) Let \( T_{\text{post}} \) be the tuple space of relation \( \text{post} \). The following function \( f \) is a score function of the tuple space \( T_{\text{post}} \):

\[
f : T_{\text{post}} \to \mathbb{R}, \mu \mapsto \mu[\text{likes}]^2 - \mu[\text{dislikes}]^2
\]

The score function \( f \) can also be expressed as the combination of the attribute score function \( s : \mathbb{N} \to \mathbb{R}, x \mapsto x^2 \) and the combining function \( c : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto x - y \) as follows:

\[
f : T \to \mathbb{R}, \mu \mapsto c(s(\mu[\text{likes}]), s(\mu[\text{dislikes}]))
\]

If the score function \( f \) is an injective function, the function \( f \) defines a total order.

Proposition 14 (Order by condition of injective score function) Let \( a_1, \ldots, a_n \) be attributes. Let \( T := \text{dom}(a_1) \times \text{dom}(a_n) \) be a tuple space. Let \( f : T \to \mathbb{R} \) be an injective tuple score function. And let \( \lambda \) be the last loaded tuple. Then, the \textit{order by} \( O = ((f((a_1, \ldots, a_n)), \uparrow)) \) defines a total order. And the \textit{order by} condition can be stated as follows:

\[
\overset{\leq \lambda}{\circ} := f((a_1, \ldots, a_n)) \leq f(\lambda)
\]
4.1. Segment Definition

Please note that the expression \( f((a_1, \ldots, a_n)) \) can only be formulated if the definition of function \( f \) is known to the top-k semantic cache, e.g., if the definition of the function \( f \) is given in the order by of the given query. The function \( f \) cannot be a black box function. Furthermore, all operations that are used in the calculation of function \( f \) must be supported by the parser and meta data component of the top-k semantic cache and also by the SMT solver.

On the other hand, if the score function is not an injective function, the function \( f \) alone does not define a total order. But we can add the primary key of the considered relation to the order by to attain a total order.

**Proposition 15 (Order by condition with arbitrary score function)** Let \( a_1, \ldots, a_n \) be attributes. Let \( T := \text{dom}(a_1) \times \text{dom}(a_n) \) be a tuple space. Let \( f : T \rightarrow \mathbb{R} \) be an arbitrary score function. And let \( \lambda \) be the last loaded tuple. In addition, let \( \{p_1, \ldots, p_m\} \subseteq \{a_1, \ldots, a_n\} \) be a primary key of the considered relation. Then the order by \( O = ((f((a_1, \ldots, a_n)), \uparrow), (p_1, \uparrow), \ldots, (p_m, \uparrow)) \) defines a total order. And the order by condition can be calculated with the help of algorithm conditionOrderBy\((O, \lambda)\) (see algorithm 2) as follows:

\[
\hat{o}^{\leq \lambda} := (f((a_1, \ldots, a_n)) < f(\lambda)) \\
\lor ((f((a_1, \ldots, a_n)) = f(\lambda)) \land \text{conditionOrderBy}(((p_1, \uparrow), \ldots, (p_m, \uparrow)), \lambda))
\]

**Example 16 (Order by condition with arbitrary score function)** Consider the score function \( f \) on relation post of example 15. The score function \( f \) is obviously not injective, because, for example, it maps all tuples that have the same number of likes and dislikes to 0. Hence, we have to add the primary key \( \{\text{id}\} \) to create an order by \( O \) that defines a total order:

\[
O = ((f((\ldots, \text{likes}, \text{dislikes}, \ldots)), \uparrow), (\text{id}, \uparrow))
\]

Using the example of last loaded tuple \( \lambda[\text{id}, \text{likes}, \text{dislikes}] = (19, 200, 100) \), we calculate the order by condition \( \hat{o}^{\leq \lambda} \) as follows:

\[
\hat{o}^{\leq \lambda} := (f((\text{id}, \ldots, \text{likes}, \text{dislikes}, \ldots)) < f(\lambda)) \\
\lor ((f((\text{id}, \ldots, \text{likes}, \text{dislikes}, \ldots)) = f(\lambda)) \land \text{conditionOrderBy}(((\text{id}, \uparrow)), (19, \ldots, 200, \text{id}, 100, 100, \ldots))) \\
\lor ((f((\text{id}, \ldots, \text{likes}, \text{dislikes}, \ldots)) = f((19, \ldots, 200, \text{id}, 100, 100, \ldots))) \\
\land \text{conditionOrderBy}(((\text{id}, \uparrow)), (19, \ldots, 200, \text{id}, 100, 100, \ldots)))
\]

\[
\lor ((\text{likes}^2 - \text{dislikes}^2 < 200^2 - 100^2) \\
\lor ((\text{likes}^2 - \text{dislikes}^2 = 200^2 - 100^2) \\
\land \text{conditionOrderBy}(((\text{id}, \uparrow)), (19, \ldots, 200, 100, 100, \ldots)))
\]

\[
\lor ((\text{likes}^2 - \text{dislikes}^2 < 30000) \lor ((\text{likes}^2 - \text{dislikes}^2 = 30000) \land (\text{id} \leq 19))
\]
4. Segments

We conclude that as long as the definition of the score function is known to the cache and the parser and meta data component of the cache as well as the utilized SMT solver support the operations of the used score function, the top-k semantic cache can support arbitrary user-defined score functions.
4.2. Set Operations on Segments

Consider a new query \( q_{\text{new}} \), which the top-k semantic cache is currently computing the answer to, whose segment \( s_{\text{new}} \) would overlap with segments \( s_1, \ldots, s_n \) of the cache. By definition, all segments of a semantic cache must be disjoint. Therefore, we need to adjust the segments. There are several ways to change the segments to achieve disjointness. The way the segments are changed, though, determines the granularity of the segments. Thus, the chosen approach has a significant effect on the performance of the semantic cache. Principally, the alternative approaches are the following:

1. **Definition 16 (Full coalescing)** \cite{JAT06} Create a new segment \( s_{\text{new}} \) for the new query \( q_{\text{new}} \). Subtract the new segment from all overlapping segments, i.e., \( \forall i \in \{1, \ldots, n\} : s_i := s_i \setminus s_{\text{new}} \). This approach is known as full coalescing.

2. **Definition 17 (No coalescing)** \cite{JAT06} Create a segment for the remainder of the new query, i.e., \( s_r := ((s_{\text{new}} \setminus s_1) \setminus \ldots) \setminus s_n = s_{\text{new}} \setminus \left( \bigcup_{i \in \{1, \ldots, n\}} s_i \right) \), for each overlap \( s_{\text{new} \cap i} := s_{\text{new}} \cap s_i \ (i \in \{1, \ldots, n\}) \), and subtract the new segment from all overlapping segments \( s_1, \ldots, s_n \), i.e., \( \forall i \in \{1, \ldots, n\} : s_i := s_i \setminus s_{\text{new}} \). This approach is known as no coalescing.

3. **Definition 18 (Total union)** Create a segment that combines the segment \( s_{\text{new}} \) for the new query and all overlapping segments \( s_1, \ldots, s_n \), i.e., create a new segment \( s_{\text{new}} \cup \left( \bigcup_{i \in \{1, \ldots, n\}} s_i \right) \).

In addition, combinations of the different approaches are, of course, also possible. The course of action for a specific query should be decided by the cache management based on a cost analysis. To allow for the use of the different strategies, a semantic cache needs to support the calculation of the difference, intersection and union of two segments. And, certainly, the semantic cache must be able to create new segments.

4.2.1. Creation of a New Segment

The creation of a new segment happens in two steps. First, an empty new segment \( s_{\text{new}} \) is created by algorithm `segmentCreate(q)` (see algorithm \[3\]). Secondly, all tuples in common with overlapping segments \( s_1, \ldots, s_n \) are moved into the new segment \( s_{\text{new}} \) by algorithm `segmentInitialInsert(s_{\text{new}}, \{s_1, \ldots, s_n\})` (see algorithm \[4\]).

The insertion of the tuples from the overlaps of all overlapping segments \( s_1, \ldots, s_n \) into the new segment \( s_{\text{new}} \) (see algorithm \[4\]) will only work, if the following two requirements are fulfilled:

\[
\bigcup_{i=1}^{n} (A_i^\text{where} \cup A_i^\text{orderby} \cup A_i^\text{initial}) \subseteq A_{\text{new}}^\text{from}
\]
4. Segments

Algorithm 3 segmentCreate(q)

**Input:** query \( q = (A_{\text{select}}, A_{\text{from}}, A_{\text{where}}, A_{\text{orderby}}, R, \hat{q}, O, k) \) (see definition 8)

**Output:** new segment \( q_{\text{new}} \)

1. \( A_{\text{select}} := A_{\text{select}} \)
2. \( A_{\text{from}} := A_{\text{from}} \)
3. \( A_{\text{where}} := A_{\text{where}} \)
4. \( A_{\text{orderby}} := A_{\text{orderby}} \)
5. \( A_{\text{initial}} := \emptyset \)
6. \( R_{\text{new}} := R \)
7. \( \hat{q}_{\text{new}} := \hat{q} \)
8. \( O_{\text{new}} := O \)
9. \( \hat{i}_{\text{new}} := \text{false} \)
10. \( \lambda_{\text{new}} := \perp \)
11. \( c_{\text{new}} := \text{false} \)
12. \( T_{\text{new}} := \emptyset \)
13. \( k_{\text{new}} := 0 \)
14. **return** \( q_{\text{new}} \)

Algorithm 4 segmentInitialInsert(\( s_{\text{new}}, \{s_1, \ldots, s_n\} \))

**Input:** \( s_{\text{new}}, s_1, \ldots, s_n \)

with \( \bigcup_{i=1}^{n} (A_i_{\text{where}} \cup A_i_{\text{orderby}} \cup A_i_{\text{initial}}) \subseteq A_{\text{from}} \)

and \( A_{\text{where}} \subseteq \bigcap_{i=1}^{n} (A_i_{\text{select}}) \)

**Output:** \( s_{\text{new}} \)

1. \( A_{\text{initial}} := \bigcup_{i=1}^{n} (A_i_{\text{where}} \cup A_i_{\text{orderby}} \cup A_i_{\text{initial}}) \)
2. \( \hat{i}_{\text{new}} := \bigvee_{i=1}^{n} ((\hat{i}_i \lor \hat{o}_i^{\leq \lambda}) \land \hat{s}_i) \)
3. \( T_{\text{new}} := \bigcup_{i=1}^{n} (\pi_{A_{\text{select}}} (\sigma_{s_{\text{new}}} (T_i))) \)
4. \( k_{\text{new}} := |T_{\text{new}}| \)
5. **return** \( s_{\text{new}} \)
2. \( A_{\text{where}}^{\text{new}} \subseteq \bigcap_{i=1}^{n} (A_{\text{select}}^i) \)

The first requirement ensures that all columns of the expression \( \hat{s}_{\text{new}} \) after execution of algorithm segmentInitialInsert(\( s_{\text{new}}, \{s_1, \ldots, s_n\} \)) are contained in the base tables \( R_{\text{new}} \) of segment \( s_{\text{new}} \). If that was not the case, we would not be able to transfer additional tuples from the server, because the server would not be able to answer the remainder query, which would use attributes that are not in the referenced tables. This requirement will be trivially satisfied if we structure the segments into equivalence classes according to the set of tables that they use.

The second requirement is necessary because we need to be able to identify the tuples of the segments \( s_1, \ldots, s_n \) that are in the overlap with segment \( s_{\text{new}} \). The tuples that need to be moved into \( s_{\text{new}} \) will be identified by evaluating the where clause \( \hat{s}_{\text{new}} \) on all overlapping segments. This will only be possible if all attributes that are used in \( \hat{s}_{\text{new}} \) are present in all overlapping segments.

The algorithm sets \( \hat{i}_{\text{new}} \) to an expression that describes all tuples of all overlapping queries (see proposition [10]) and updates the attribute set \( A_{\text{initial}}^{\text{new}} \). Please note that we do not need a constraint that specifies that we only mean tuples that also satisfy \( \hat{s}_{\text{new}} \), because this is known implicitly by definition of \( \hat{i}_{\text{new}} \). In addition, the algorithm inserts all tuples from segments \( s_1, \ldots, s_n \) that satisfy \( \hat{s}_{\text{new}} \) into segment \( s_{\text{new}} \) and updates \( k_{\text{new}} \) accordingly.

In combination, algorithm segmentCreate(\( q \)) and algorithm segmentInitialInsert(\( s_{\text{new}}, \{s_1, \ldots, s_n\} \)) allow us to create new segments for the top-k semantic cache. Please note that even after execution of these algorithms, the new segments \( s_{\text{new}} \) overlaps with segments \( s_1, \ldots, s_n \). We still need to subtract the new segment from all overlapping segments.

The expression that describes all loaded tuples of a segment \( s \), i.e., \( (\hat{\sigma} \leq \lambda \lor \hat{i}) \land \hat{s} \), can be quite complex, especially, if tuples from many other segments have been moved into the segment \( s \) when it was created. The expression becomes much simpler, namely just \( \hat{s} \), when a segment is completely loaded.

---

**Algorithm 5** segmentComplete(\( s \))

**Input:** \( s \)

**Output:** \( s \)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A_{\text{orderby}} := \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( A_{\text{initial}} := \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>( O := () )</td>
</tr>
<tr>
<td>4</td>
<td>( \hat{i} := \text{false} )</td>
</tr>
<tr>
<td>5</td>
<td>( \lambda := \bot )</td>
</tr>
<tr>
<td>6</td>
<td>( c := \text{true} )</td>
</tr>
<tr>
<td>7</td>
<td><strong>return</strong> ( s )</td>
</tr>
</tbody>
</table>
4. Segments

To be able to use this optimization, the algorithm \texttt{segmentComplete(\(s\))} (see algorithm 5) will be executed when a segment has been completely loaded. If the segment has been completely loaded, the order is irrelevant, because the segment now supports any order. Therefore, \(O\) is set to the empty list, \(\mathcal{A}_{\text{orderby}}\) is set to the empty set, and \(\lambda\) is set to \(\bot\), because the last loaded tuple is not needed any more. In addition, we no longer need to know which tuples have been initially inserted into the segment originating in other segments. Therefore, we set \(\hat{i}\) to false and update \(\mathcal{A}_{\text{initial}}\) to the empty set.

**Example 17 (Segment \(s_2\) for query \(q_2\))** Consider query \(q_2\) (see listing 4.2), the second query of the running example (see section 1.2):

```sql
1 SELECT p.id, p.likes, p.created, p.title, p.author
2 FROM post p
3 WHERE (p.created >= TODAY)
4 AND ((p.created >= LAST_HOUR) OR (p.likes >= 20))
5 ORDER BY p.id ASC
6 LIMIT 10
```

Listing 4.2: Query \(q_2\): a top-k query returning today’s posts by id

As a start, we create a new and initially empty segment \(s_2\) for query \(q_2\) using the algorithm \texttt{segmentCreate(\(q_2\))} (see algorithm 3). The result of this algorithm is shown in table 4.6.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A^\text{select}) (_2)</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>(A^\text{from}) (_2)</td>
<td>All attributes of relation \textit{post}</td>
</tr>
<tr>
<td>(A^\text{where}) (_2)</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>(A^\text{orderby}) (_2)</td>
<td>{id}</td>
</tr>
<tr>
<td>(A^\text{initial}) (_2)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R^2)</td>
<td>{post}</td>
</tr>
<tr>
<td>(\hat{s}^2)</td>
<td>(created (\geq) today) (\land) ((created (\geq) last_hour) (\lor) (likes (\geq) 20))</td>
</tr>
<tr>
<td>(\hat{O}^2)</td>
<td>((id, (\uparrow)))</td>
</tr>
<tr>
<td>(k^2)</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\hat{i}^2)</td>
<td>false</td>
</tr>
<tr>
<td>(\lambda^2)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>(c^2)</td>
<td>false</td>
</tr>
<tr>
<td>(T^2)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

Table 4.6.: Values of empty segment \(s_2\) created by \texttt{segmentCreate(\(q_2\))} (see algorithm 3)

We assume that query \(q_1\) has been executed before query \(q_2\). A cache segment \(s_1\) has been created for query \(q_1\) (see example 11). Segment \(s_1\) and segment \(s_2\) overlap....
4.2. Set Operations on Segments

because the following expression is satisfiable:

\[
\hat{s}_1 \land \hat{s}_2 = ((\text{created} \geq \text{last week}) \land (\text{likes} \geq 100)) \\
\land ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last hour}) \lor (\text{likes} \geq 20)))
\]

This can be determined by an SMT solver (see proposition 7). Therefore, we use algorithm segmentInitialInsert(\(s_2, \{s_1\}\)) (see algorithm 4) to insert the tuples that are contained in the overlap with the overlapping segment \(s_1\) into segment \(s_2\).

The algorithm segmentInitialInsert(\(s_2, \{s_1\}\)) consists of four steps: First, it calculates \(A_{\text{initial}}\), the set of all attributes that are used in \(\hat{i}_2\).

\[
A_{\text{initial}} := A_{\text{where}} \cup A_{\text{orderby}} \cup A_{\text{initial}}
\]

\[
= \{\text{created}, \text{likes}\} \cup \{\text{likes}\} \cup \{}
\]

\[
= \{\text{created}, \text{likes}\}
\]

Secondly, the algorithm segmentInitialInsert(\(s_2, \{s_1\}\)) determines \(\hat{i}_2\). The expression \(\hat{i}_2\) describes all tuples that are inserted into segment \(s_2\) by this algorithm. The tuples that have been loaded by segment \(s_1\) are described by \(\hat{s}_1 \land (\hat{o}_{\text{i}} \lor \hat{i}_1)\) (see proposition 10). Example 10 shows expression \(\hat{s}_1\):

\[
\hat{s}_1 = (\text{created} \geq \text{last week}) \land (\text{likes} \geq 100)
\]

And expression \(\hat{o}_{\text{i}} \lor \hat{i}_1\) is calculated in example 13:

\[
\hat{o}_{\text{i}} \lor \hat{i}_1 = (\text{likes} \equiv \bot) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))
\]

The expression \(\hat{i}_1 = \text{false}\) (see example 11). Hence, we calculate \(\hat{s}_1 \land (\hat{o}_{\text{i}} \lor \hat{i}_1) =: \hat{i}_2\) as follows:

\[
\hat{i}_2 := \hat{s}_1 \land (\hat{o}_{\text{i}} \lor \hat{i}_1)
\]

\[
= \hat{s}_1 \land \hat{o}_{\text{i}}
\]

\[
= ((\text{created} \geq \text{last week}) \land (\text{likes} \geq 100)) \\
\land ((\text{likes} \equiv \bot) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))
\]

\[
\equiv (\text{created} \geq \text{last week}) \land ((\text{likes} \equiv \bot) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))
\]

Recall that we have defined the expression \(\hat{i}_2\) such that the restriction to segment \(s_2\) is implicit, i.e., that the expression \(\hat{s}_2\) does not need to contain the constraint \(\hat{s}_2\) (see section 4.1.2).
4. Segments

Thirdly, the algorithm `segmentInitialInsert(s_2, \{s_1\})` updates \(T_2\), i.e., all tuples of overlaps are copied into segment \(s_2\) (see figure 4.4, table 4.7, and table 1.1):

\[
T_2 := \pi_{A_2}^{\pi_{A_2}}(\sigma_{\hat{s}_2}(T_1)) = \{\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}\}
\]

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>(q_1)</th>
<th>(q_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>(\mu_7)</td>
<td>7</td>
<td>220</td>
<td>(\text{today})</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>(\mu_8)</td>
<td>8</td>
<td>180</td>
<td>(\text{today})</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>(\mu_9)</td>
<td>9</td>
<td>170</td>
<td>(\text{last hour})</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>(\mu_{10})</td>
<td>10</td>
<td>230</td>
<td>(\text{last hour})</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>(\mu_{14})</td>
<td>14</td>
<td>190</td>
<td>(\text{today})</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>(\mu_{19})</td>
<td>19</td>
<td>200</td>
<td>(\text{last hour})</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4.7.: Result of \(\pi_{A_2}^{\pi_{A_2}}(\sigma_{\hat{s}_2}(T_1))\)

At last, the number of tuples \(k_2\) of segment \(s_2\) is set.

\[
k_2 := |T_2| = |\{\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}\}| = 6
\]

Segment \(s_2\) after the successful execution of algorithm `segmentInitialInsert(s_2, \{s_1\})` is shown in table 4.8.

Finally, we have to load the remaining tuples that are needed to answer query \(q_2\), but which are not contained in the top-k semantic cache, from the server. To do this, we write a *remainder query* \(r_2\) that retrieves the missing tuples (see section 6.2.4):

\[
r_2 := q_2 \setminus (\pi_{A_2}^{\pi_{A_2}}(\sigma_{\hat{s}_2}(\times_{r \in R}(r))))
\]

In SQL, the remainder query \(r_2\) can be stated as follows:

```sql
```
4.2. Set Operations on Segments

Figure 4.4.: Creating a segment $s_2$ for query $q_2$ using segment $s_1$ of query $q_1$
4. Segments

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^\text{select}_2$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A^\text{from}_2$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A^\text{where}_2$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A^\text{orderby}_2$</td>
<td>{id}</td>
</tr>
<tr>
<td>$A^\text{initial}_2$</td>
<td>{created, likes}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2_2$</td>
<td>(post)</td>
</tr>
<tr>
<td>$s^2_2$</td>
<td>(created $\geq$ today) $\land$ ((created $\geq$ last_hour) $\lor$ (likes $\geq$ 20))</td>
</tr>
<tr>
<td>$O^2_2$</td>
<td>((id, ↑))</td>
</tr>
<tr>
<td>$k^2_2$</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{i}^2_2$</td>
<td>(created $\geq$ last_week) $\land$ ((likes $&gt;$ 160) $\lor$ ((likes = 160) $\land$ (id $\leq$ 2)))</td>
</tr>
<tr>
<td>$\lambda^2_2$</td>
<td>false</td>
</tr>
<tr>
<td>$c^2_2$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$T^2_2$</td>
<td>${\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}}$</td>
</tr>
</tbody>
</table>

Table 4.8.: Values of segment $s^2_2$ after the insertion of tuples from overlaps

Listing 4.3: Query $r^2_2$: remainder query for segment $s^2_2$

The processing overhead at the server that has to evaluate the remainder query $r^2_2$ instead of query $q^1_1$ is negligibly small. The database system can just calculate the result of query $q^2_2$ as before. Then, it filters the result, which can be pipelined, and only transfers the tuples to the client that do not satisfy the condition $\hat{i}^2_2$.

Some databases, e.g., PostgreSQL, might not be able to generate an efficient query execution plan based on this remainder query. We can help the database by reformulating the remainder query in the following way:

$$r'_2 := \sigma_{\neg \hat{i}^2_2}(q^2_2)$$

In SQL, the remainder query $r'_2$ can be stated as follows:

```sql
SELECT s.id, s.likes, s.created, s.title, s.author
FROM (SELECT p.id, p.likes, p.created, p.title, p.author
FROM post p
WHERE (p.created >= TODAY)
AND ((p.likes > 160) OR ((p.likes = 160) AND (p.id <= 2)))
```

WHERE (p.created $\geq$ LAST_WEEK)

AND ((p.likes > 160) OR ((p.likes = 160) AND (p.id <= 2)))

)
4.2. Set Operations on Segments

Listing 4.4: Query $r'_2$: rewritten remainder query for segment $s_2$

For query $r_2$, PostgreSQL generates a query execution plan that contains a sequential scan on relation post to execute sub-query $\left(\pi_{A}^{\text{select}}(\sigma_{r_2}^{\text{except}}(r_2'))\right)$. In contrast, for query $r'_2$, PostgreSQL creates a query execution plan that just filters the result of query $q$ with the filter $\neg \hat{i} \lor u\hat{i}$ (see table 4.9).

<table>
<thead>
<tr>
<th>Query execution plan of $r_2$</th>
<th>Query execution plan of $r'_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashSetOp Except</td>
<td>Subquery Scan on s</td>
</tr>
<tr>
<td>→ Append</td>
<td>Filter: $\neg \hat{i} \lor u\hat{i}$</td>
</tr>
<tr>
<td>→ Subquery Scan</td>
<td>→ Limit</td>
</tr>
<tr>
<td>→ Limit</td>
<td>→ Index Scan on post</td>
</tr>
<tr>
<td>→ Index Scan on post</td>
<td>Filter: $\hat{q}$</td>
</tr>
<tr>
<td>→ Subquery Scan</td>
<td>→ Seq Scan on post</td>
</tr>
<tr>
<td>→ Subquery Scan</td>
<td>Filter: $\hat{i}$</td>
</tr>
<tr>
<td>→ Seq Scan on post</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9.: Query execution plans of $r_2$ and $r'_2$ in PostgreSQL

The result of the remainder query $r_2$ is shown in table 4.10.

<table>
<thead>
<tr>
<th>post ID</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>...</th>
<th>$q_1$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>$\mu_{11}$</td>
<td>11</td>
<td>20</td>
<td>today</td>
<td>...</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>$\mu_{12}$</td>
<td>12</td>
<td>40</td>
<td>last hour</td>
<td>...</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>$\mu_{13}$</td>
<td>13</td>
<td>80</td>
<td>last hour</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>$\mu_{15}$</td>
<td>15</td>
<td>50</td>
<td>today</td>
<td>...</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>$\mu_{16}$</td>
<td>16</td>
<td>140</td>
<td>last hour</td>
<td>...</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.10.: Result of $r_2 := q_2 \setminus (\pi_{A}^{\text{select}}(\sigma_{r_2}^{\text{except}}(r_2')))$
4. Segments

It is added to the tuples $T_2$ of segment $s_2$:

\[
T_2 := T_2 \cup \{ \mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16} \} = \{ \mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{11}, \mu_{12}, \mu_{13}, \mu_{14}, \mu_{15}, \mu_{16}, \mu_{19} \}
\]

And the number of tuples $k_2$ in segment $s_2$ is updated accordingly:

\[
k_2 := |T_2| = |\{ \mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{11}, \mu_{12}, \mu_{13}, \mu_{14}, \mu_{15}, \mu_{16}, \mu_{19} \}| = 11
\]

In addition, $\lambda_2$, which contains the tuple that has been loaded last, is set to $\mu_{16}$ as it has been loaded last by the remainder query $r_2$ (see table 4.10). Table 4.11 and figure 4.5 display the final state of segment $s_2$ after completed processing of query $q_2$.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_2^\text{select}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_2^\text{from}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_2^\text{where}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_2^\text{orderby}$</td>
<td>{id}</td>
</tr>
<tr>
<td>$A_2^\text{initial}$</td>
<td>{created, likes}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_2$</td>
<td>{post}</td>
</tr>
<tr>
<td>$s_2$</td>
<td>(created $\geq$ today) $\land$ ((created $\geq$ last hour) $\lor$ (likes $\geq$ 20))</td>
</tr>
<tr>
<td>$O_2$</td>
<td>((id, ↑))</td>
</tr>
<tr>
<td>$k_2$</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{\iota}_2$</td>
<td>(created $\geq$ last week) $\land$ ((likes $&gt; 160$) $\lor$ ((likes $= 160$) $\land$ (id $\leq 2$)))</td>
</tr>
<tr>
<td>$\lambda_2$</td>
<td>$\mu_{16}[\text{id, likes, created}] = (16, 140, \text{last hour})$ (see table 4.10)</td>
</tr>
<tr>
<td>$c_2$</td>
<td>false</td>
</tr>
<tr>
<td>$T_2$</td>
<td>${ \mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{11}, \mu_{12}, \mu_{13}, \mu_{14}, \mu_{15}, \mu_{16}, \mu_{19} }$</td>
</tr>
</tbody>
</table>

Table 4.11.: Values of segment $s_2$ after completed processing of query $q_2$

Please note that after execution of segmentCreate($q_2$) and segmentInitialInsert($s_2, \{ s_1 \}$), the new segments $s_2$ still overlaps with segment $s_1$. We still need to subtract the new segment $s_2$ from the overlapping segment $s_1$. 76
4.2. Set Operations on Segments

Figure 4.5.: Segment $s_2$ for query $q_2$
4. Segments

4.2.2. Difference of Two Segments

Since all segments of a semantic cache are disjoint, we only ever need to calculate the difference of an existing segment \( s_c \) and the newly created segment \( s_{\text{new}} \) of a query \( q_{\text{new}} \) (see figure 4.6). Algorithm `segmentDifferenceInPlace(\( s_c, s_{\text{new}} \))` can calculate this difference in-place (see algorithm 6).

![Figure 4.6.: Difference of two segments](image)

**Algorithm 6** \( s_c \setminus s_{\text{new}} \): `segmentDifferenceInPlace(\( s_c, s_{\text{new}} \))`

**Input:** \( s_c, s_{\text{new}} \) with \( A_{\text{where}}^{\text{new}} \subseteq A_{\text{select}} \)

**Output:** \( s_c := s_c \setminus s_{\text{new}} \)

1. \( A_{\text{where}} := A_{\text{where}}^{\text{new}} \cup A_{\text{where}} \)
2. \( \hat{s}_c := \hat{s}_c \land (\neg \hat{s}_{\text{new}} \lor \mu \hat{s}_{\text{new}}) \)
3. \( T_c := \sigma_{\neg \hat{s}_{\text{new}} \lor \mu \hat{s}_{\text{new}}}(T_c) \)
4. \( k_c := |T_c| \)
5. return \( s_c \)

The algorithm requires that all attributes of the new segment are contained in segment \( s_c \), i.e., \( A_{\text{where}}^{\text{new}} \subseteq A_{\text{select}} \). This requirement is necessary, because we will need to remove all tuples from the segment that satisfy the `where` clause \( \hat{s}_{\text{new}} \) of \( s_{\text{new}} \). To identify these tuples, we need to be able to evaluate \( \hat{s}_{\text{new}} \neq \text{true} \equiv (\neg \hat{s}_{\text{new}} \lor \mu \hat{s}_{\text{new}}) \). This will only be possible, if the attributes that are used in \((\neg \hat{s}_{\text{new}} \lor \mu \hat{s}_{\text{new}})\) are present in the segment. Please note that this requirement is essentially the same one as the requirement for initial insertion of tuples into newly created segments in algorithm `segmentInitialInsert(s_{\text{new}}, \{s_1, \ldots, s_n\})` (see algorithm 4). Therefore, if a segment satisfies this requirement, we will be able to both insert the tuples from the overlap into the newly created segment and remove the tuples from the overlapping segment of the cache.
The algorithm sets the condition of segment \( s_c \) to 
\[
\hat{s}_c \land (\neg \hat{s}_{\text{new}} \lor u\hat{s}_{\text{new}})
\]
and updates the attribute set \( A_{\text{new}}^\text{where} \) by adding all attributes of \( \hat{s}_{\text{new}} \) to it. Furthermore, all tuples that no longer satisfy the updated segment condition \( \hat{s}_c \) are removed by filtering the tuples of the segment with the negation of the \textit{where} clause of \( s_{\text{new}} \), i.e., 
\[
T_c := T_c \setminus \sigma_{\hat{s}_{\text{new}}} (T_c) \text{ or equivalently } T_c := \sigma_{\neg \hat{s}_{\text{new}} \lor u\hat{s}_{\text{new}}} (T_c).
\]
Finally, \( k_c \) is updated to reflect the new number of tuples of segment \( s_c \).

Example 18 (Difference of segment \( s_1 \) and segment \( s_2 \)) In example [17], we have created a new segment \( s_2 \) for query \( q_2 \). Because segment \( s_2 \) overlapped with segment \( s_1 \), we have copied the tuples that were contained in the overlap with segment \( s_2 \) to segment \( s_2 \). But we have not yet removed these tuples from segment \( s_1 \). We have not updated it accordingly to restore the disjointness property of the top-k semantic cache.

Hence, we will now update segment \( s_1 \) with algorithm \text{segmentDifferenceInPlace}(s_1, s_2) \) (see algorithm [6]).

To begin with, the algorithm modifies \( A_{\text{where}}^1 \) by adding all attributes of \( A_{\text{where}}^2 \) to it:
\[
A_{\text{where}}^1 := A_{\text{where}}^1 \cup A_{\text{where}}^2 = \{\text{created, likes}\} \cup \{\text{created, likes}\} = \{\text{created, likes}\}
\]

Then, it determines the new segment condition \( \hat{s}_1 \):
\[
\hat{s}_1 := \hat{s}_1 \land (\neg \hat{s}_2 \lor u\hat{s}_2) = (\text{created} \geq \text{last_week}) \land (\text{likes} \geq 100) \land \\
\neg((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20))) \\
\lor u((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20)))
\]
\[
\equiv \text{true} \ (\text{created} \geq \text{last_week}) \land (\text{likes} \geq 100) \land \\
\neg((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20))) \\
\equiv (\text{created} \geq \text{last_week}) \land (\text{likes} \geq 100) \land \\
((\text{created} < \text{today}) \lor \neg((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20))) \\
\equiv (\text{created} \geq \text{last_week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)
\]

Afterward, the set of tuples \( T_1 \) of segment \( s_1 \) is reduced to the tuples that satisfy the new segment condition \( \hat{s}_1 \), i.e., all tuples that have been copied to segment \( s_2 \) are removed from segment \( s_1 \) (see figure [4.7]):
\[
T_1 := \sigma_{\neg \hat{s}_2 \lor u\hat{s}_2} (T_1) = \{\mu_1, \mu_2, \mu_3, \mu_4\}
\]
Figure 4.7.: Segment $s_1$ after creation and subtraction of segment $s_2$
Finally, the number of tuples $k_1$ in segment $s_2$ is updated accordingly:

$$k_1 := |T_1| = |\{\mu_1, \mu_2, \mu_3, \mu_4\}| = 4$$

Table 4.12 shows segment $s_1$ after completion of algorithm segmentDifferenceInPlace($s_1$, $s_2$) (see algorithm 6).

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{select}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_{from}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_{where}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_{orderby}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_{initial}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>{post}</td>
</tr>
<tr>
<td>$\hat{s}_1$</td>
<td>{(created $\geq$ last week) $\land$ (created $&lt;$ today) $\land$ (likes $\geq$ 100)}</td>
</tr>
<tr>
<td>$O_1$</td>
<td>$((\text{likes}, \downarrow), (\text{id}, \uparrow))$</td>
</tr>
<tr>
<td>$k_1$</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{i}_1$</td>
<td>false</td>
</tr>
<tr>
<td>$\lambda_1$</td>
<td>$\mu_1</td>
</tr>
<tr>
<td>$c_1$</td>
<td>false</td>
</tr>
<tr>
<td>$T_1$</td>
<td>{\mu_1, \mu_2, \mu_3, \mu_4}</td>
</tr>
</tbody>
</table>

Table 4.12.: Values of segment $s_1$ after subtraction of segment $s_2$

**Example 19 (Segment $s_3$ for query $q_3$)** Consider query $q_3$ (see listing 4.5), the third query of the running example (see section 1.2):

```
1 SELECT p.id, p.likes, p.created, p.title, p.author
2 FROM post p
3 WHERE p.created >= LAST_HOUR
4 ORDER BY p.likes DESC
5 LIMIT 10
```

Listing 4.5: Query $q_3$: a top-k query returning the most liked posts of the last hour

Again, we start off by creating a new and initially empty segment $s_3$ for query $q_3$ using the using the algorithm segmentCreate($q_3$) (see algorithm 3). Table 4.13 shows the resulting segment $s_3$.

We assume that the top-k semantic cache currently contains two segments, segment $s_1$ (see example 18) and segment $s_2$ (see example 17).
### 4. Segments

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_3^{\text{select}}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_3^{\text{from}}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_3^{\text{where}}$</td>
<td>{created}</td>
</tr>
<tr>
<td>$A_3^{\text{orderby}}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_3^{\text{initial}}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_3$</td>
<td>(post)</td>
</tr>
<tr>
<td>$s_3$</td>
<td>created $\geq$ last_hour</td>
</tr>
<tr>
<td>$O_3$</td>
<td>(likes ↓, id ↑)</td>
</tr>
<tr>
<td>$k_3$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{i}_3$</td>
<td>false</td>
</tr>
<tr>
<td>$\lambda_3$</td>
<td>$\perp$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>false</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Table 4.13.: Values of empty segment $s_3$ created by segmentCreate($q_3$) (see algorithm 3)

Segment $s_3$ and segment $s_1$ do not overlap because the following expression is clearly not satisfiable:

$$\hat{s}_3 \land \hat{s}_1 = (\text{created} \geq \text{last\_hour})$$

$$\land ((\text{created} \geq \text{last\_week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100))$$

The sub-expression $(\text{created} \geq \text{last\_hour}) \land (\text{created} < \text{today})$ cannot be evaluated to true by assignment of any values. Therefore, we need not consider segment $s_1$ any further. This can be determined by an SMT solver (see proposition 7).

Please note that, initially, before the subtraction of segment $s_2$ from segment $s_1$, segment $s_3$ and segment $s_1$ did overlap (see figure 1.6 in section 1.2). The conjunction of their segment conditions, i.e., $\hat{s}_3 \land \hat{s}_1^{\text{old}}$, is satisfiable:

$$\hat{s}_3 \land \hat{s}_1^{\text{old}} = (\text{created} \geq \text{last\_hour})$$

$$\land (\text{created} \geq \text{last\_week}) \land (\text{likes} \geq 100)$$

Segment $s_3$ and segment $s_2$, however, do overlap. The expression $\hat{s}_3 \land \hat{s}_2$ is satisfiable:

$$\hat{s}_3 \land \hat{s}_2 = (\text{created} \geq \text{last\_hour})$$

$$\land ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20)))$$

In fact, it can be shown that segment $s_2$ subsumes segment $s_3$, i.e., segment $s_3$ is completely contained in segment $s_2$, because $(\text{created} \geq \text{last\_hour}) \rightarrow ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20))$.
4.2. Set Operations on Segments

today) ∧ ((created ≥ last_hour) ∨ (likes ≥ 20))) is true for any tuple (see definition 7). Once again, this can be determined by an SMT solver (see proposition 8). Because segment $s_3$ and segment $s_2$ overlap, we use algorithm `segmentInitialInsert($s_3$, $\{s_2\}$)` (see algorithm 4) to copy the tuples that are contained in the overlap with segment $s_2$ to segment $s_3$.

First, the algorithm updates $A_{3}^{\text{initial}}$:

$$A_{3}^{\text{initial}} := A_2^{\text{where}} \cup A_2^{\text{orderby}} \cup A_2^{\text{initial}}$$

$$= \{\text{created, likes} \} \cup \{\text{id} \} \cup \{\text{created, likes} \}$$
$$= \{\text{created, likes, id} \}$$

Secondly, condition $\hat{i}_3$ is calculated, which describes the tuples that are added to segment $s_3$ by algorithm `segmentInitialInsert($s_3$, $\{s_2\}$)` (see algorithm 4). It is defined as $\hat{i}_3 = (\hat{i}_2 \vee \hat{o}_2^{\leq \lambda}) \land \hat{s}_2$. Therefore, previously, we need to determine the expression $\hat{o}_2^{\leq \lambda}$. This expression is calculated as follows (see algorithm 2):

$$\hat{o}_2^{\leq \lambda} := \text{conditionOrderBy}(O_2, \lambda_2)$$
$$= \text{conditionOrderBy}((\text{id} \uparrow), (16, \ldots))$$
$$= \text{id} \leq 16$$

The expressions $\hat{i}_2$ and $\hat{s}_2$ are already known. We take them from the description of segment $s_2$ (see table 4.11):

$$\hat{i}_2 = (\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$$
$$\hat{s}_2 = (\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20))$$

With these expressions, we can calculate the condition $\hat{i}_3$ (see proposition 10 and algorithm 4) as follows:

$$\hat{i}_3 := (\hat{i}_2 \lor \hat{o}_2^{\leq \lambda}) \land \hat{s}_2$$
$$= ((\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))) \lor (\text{id} \leq 16))$$
$$\land ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20)))$$
$$= (((\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))) \lor (\text{id} \leq 16)))$$
$$\land (\text{created} \geq \text{today})$$
$$\land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20))$$
$$= (((\text{created} \geq \text{last\_week}) \land ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20)))$$
$$\land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)) \lor (\text{id} \leq 16))$$}
$$\land (\text{created} \geq \text{today})$$
$$\land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20))$$
$$= (((\text{created} \geq \text{last\_week}) \land ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20)))$$
$$\land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)) \lor (\text{id} \leq 16))$$}
$$\land (\text{created} \geq \text{today})$$
$$\land ((\text{created} \geq \text{last\_hour}) \lor (\text{likes} \geq 20))$$
$$= (((\text{created} \geq \text{last\_week}) \land ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20)))$$
$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16))$$
4. Segments

After this, algorithm segmentInitialInsert(s3, \{s2\}) (see algorithm 4) updates T3 by copying the tuples that are contained in the overlap of segment s3 and segment s2 to segment s3 (see figure 4.8, table 4.14, and table 1.1):

\[
T_3 := \pi_{A_3}^\text{select}(\sigma_{s_3}(T_2)) = \{\mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}\}
\]

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>(q_1)</th>
<th>(q_2)</th>
<th>(q_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(\mu_{10})</td>
<td>10</td>
<td>230 last hour</td>
<td>(\hat{3})</td>
<td>(\hat{4})</td>
<td>(\hat{1})</td>
</tr>
<tr>
<td>19</td>
<td>(\mu_{19})</td>
<td>19</td>
<td>200 last hour</td>
<td>(\hat{6})</td>
<td>(\hat{12})</td>
<td>(\hat{2})</td>
</tr>
<tr>
<td>9</td>
<td>(\mu_9)</td>
<td>9</td>
<td>170 last hour</td>
<td>(\hat{9})</td>
<td>(\hat{3})</td>
<td>(\hat{3})</td>
</tr>
<tr>
<td>16</td>
<td>(\mu_{16})</td>
<td>16</td>
<td>140 last hour</td>
<td>(\hat{12})</td>
<td>(\hat{10})</td>
<td>(\hat{4})</td>
</tr>
<tr>
<td>13</td>
<td>(\mu_{13})</td>
<td>13</td>
<td>80 last hour</td>
<td>(\hat{7})</td>
<td>(\hat{8})</td>
<td>(\hat{4})</td>
</tr>
<tr>
<td>12</td>
<td>(\mu_{12})</td>
<td>12</td>
<td>40 last hour</td>
<td>(\hat{6})</td>
<td>(\hat{6})</td>
<td>(\hat{3})</td>
</tr>
</tbody>
</table>

Table 4.14.: Result of \(\pi_{A_3}^\text{select}(\sigma_{s_3}(T_2))\)

Finally, the number of tuples in segment s3 is set accordingly:

\[
k_3 := |T_3| = |\{\mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}\}| = 6
\]

Table 4.15 shows segment s3 after execution of algorithm segmentInitialInsert(s3, \{s2\}) (see algorithm 4).

Following, we have to load the missing tuples that are needed to answer query \(q_3\), but which are not contained in the top-k semantic cache, from the server. Therefore, we need to ask a remainder query \(r_3\) that returns the remaining tuples (see section 6.2.4):

\[
r_3 := q_3 \setminus (\pi_{A_3}^\text{select}(\sigma_{s_3}(\times_{r \in R_3}r)))
\]

In SQL, the remainder query \(r_3\) can be stated as follows:

```sql
SELECT p.id, p.likes, p.created, p.title, p.author
FROM post p
WHERE p.created >= LAST_HOUR
ORDER BY p.likes DESC, p.id ASC
LIMIT 10
EXCEPT
( SELECT p.id, p.likes, p.created, p.title, p.author
FROM post p
WHERE p.created >= LAST_HOUR
ORDER BY p.likes DESC, p.id ASC
LIMIT 10
)
```
4.2. Set Operations on Segments

Figure 4.8: Creating a segment $s_3$ for query $q_3$ using segment $s_2$ of query $q_1$ and query $q_2$
4. Segments

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{\text{select}} )</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>( A_{\text{from}} )</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>( A_{\text{where}} )</td>
<td>{created}</td>
</tr>
<tr>
<td>( A_{\text{orderby}} )</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>( A_{\text{initial}} )</td>
<td>{created, likes, id}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_3 )</td>
<td>{post}</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>created ≥ last hour</td>
</tr>
<tr>
<td>( O_3 )</td>
<td>((\text{likes}, \downarrow), (\text{id}, \uparrow)))</td>
</tr>
<tr>
<td>( k_3 )</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{i}_3 )</td>
<td>(((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))) \land ((\text{likes} &gt; 160) \lor (\text{id} \leq 16)))</td>
</tr>
<tr>
<td>( \lambda_3 )</td>
<td>false</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>{(\mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19})}</td>
</tr>
</tbody>
</table>

Table 4.15.: Values of segment \( s_3 \) after the insertion of tuples from overlaps

\begin{verbatim}
11 FROM post p
12 WHERE ((p.created >= LAST_HOUR) OR ((p.created >= TODAY) AND (p.likes >= 20)))
13 AND ((p.likes > 160) OR (p.id <= 16))
\end{verbatim}

Listing 4.6: Query \( r_3 \): remainder query for segment \( s_3 \)

Table 4.16 shows the result of the remainder query \( r_3 \).

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>q_1</th>
<th>q_2</th>
<th>q_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>(\mu_{22})</td>
<td>22</td>
<td>130</td>
<td>13</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>(\mu_{21})</td>
<td>21</td>
<td>90</td>
<td>14</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>(\mu_{23})</td>
<td>23</td>
<td>70</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>(\mu_{25})</td>
<td>25</td>
<td>60</td>
<td>17</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>(\mu_{27})</td>
<td>27</td>
<td>50</td>
<td>19</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.16.: Result of \( r_3 := q_3 \setminus (\pi_{A_{\text{select}}} (\sigma_{\hat{i}_3} (\times_{r \in R_3} r)))\)
4.2. Set Operations on Segments

The result of the remainder query \( r_3 \) is added to the tuples \( T_3 \) of segment \( s_3 \):

\[
T_3 := T_3 \cup \{ \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27} \}
\]

\[
= \{ \mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27} \}
\]

And the number of tuples \( k_3 \) in segment \( s_3 \) is updated accordingly:

\[
k_3 := |T_3| = |\{ \mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27} \}| = 11
\]

In addition, \( \lambda_3 \), which contains the tuple that has been loaded last, is set to \( \mu_{27} \). This tuple is the last one that has been loaded by the remainder query \( r_2 \) (see table 4.10). Table 4.17 and figure 4.9 display the final state of segment \( s_3 \) after completed processing of query \( q_3 \).

Table 4.17.: Values of segment \( s_3 \) after completed processing of query \( q_3 \)

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_3^{select} )</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>( A_3^{from} )</td>
<td>All attributes of relation ( post )</td>
</tr>
<tr>
<td>( A_3^{where} )</td>
<td>{created}</td>
</tr>
<tr>
<td>( A_3^{orderby} )</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>( A_3^{initial} )</td>
<td>{created, likes, id}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_3 )</td>
<td>{post}</td>
</tr>
<tr>
<td>( \hat{s}_3 )</td>
<td>created ( \geq ) last hour</td>
</tr>
<tr>
<td>( O_3 )</td>
<td>((likes, ( \downarrow)), (id, ( \uparrow)))</td>
</tr>
<tr>
<td>( k_3 )</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{i}_3 )</td>
<td>((created ( \geq ) last_hour) ( \lor ) ((created ( \geq ) today) ( \land ) (likes ( \geq ) 20))) ( \land ) ((likes &gt; 160) ( \lor ) (id ( \leq ) 16))</td>
</tr>
<tr>
<td>( \lambda_3 )</td>
<td>( \mu_{27} ) [id, likes, created] = (27, 50, last_hour) (see table 4.16)</td>
</tr>
<tr>
<td>( c_3 )</td>
<td>false</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>{ \mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27} }</td>
</tr>
</tbody>
</table>

Example 20 (Difference of segment \( s_2 \) and segment \( s_3 \)) In example 19, we have created a new segment \( s_3 \) for query \( q_3 \). Because segment \( s_3 \) overlapped with segment \( s_2 \), we have copied the tuples that were contained in the overlap with segment \( s_3 \) to segment \( s_3 \). But we have not removed these tuples from segment \( s_2 \). We have not yet restored the disjointness property of the top-k semantic cache. Therefore, we must use algorithm segmentDifferenceInPlace(\( s_2, s_3 \)) (see algorithm 6) to update segment \( s_2 \).
Figure 4.9.: Segment $s_3$ for query $q_3$
First of all, the algorithm adds all attributes of \( A_3 \) to \( A_2 \):
\[
A_2 := A_2 \cup A_3 = \{\text{created, likes}\} \cup \{\text{created}\} = \{\text{created, likes}\}
\]

Secondly, the algorithm \( \text{segmentDifferenceInPlace}(s_2, s_3) \) (see algorithm 6) calculates the new segment condition \( \hat{s}_2 \):
\[
\hat{s}_2 := s_2 \land (\neg \hat{s}_3 \lor u\hat{s}_3)
\]
\[
= ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20)))
\]
\[
\land (\neg(\text{created} \geq \text{last_hour}) \lor u(\text{created} \geq \text{last_hour}))
\]
\[
\equiv \text{true} ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20)))
\]
\[
\land (\neg(\text{created} \geq \text{last_hour})
\]
\[
\equiv ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last_hour}) \lor (\text{likes} \geq 20)))
\]
\[
\land (\text{created} < \text{last_hour})
\]
\[
\equiv (\text{created} \geq \text{today}) \land (\text{created} < \text{last_hour}) \land (\text{likes} \geq 20)
\]

Thirdly, the set of tuples \( T_2 \) of segment \( s_2 \) is updated. All tuples that have been copied to segment \( s_3 \) are removed from segment \( s_2 \) (see figure 4.10):
\[
T_2 := \sigma_{\neg \hat{s}_3 \lor u\hat{s}_3}(T_2)
\]
\[
= \{\mu_7, \mu_8, \mu_{11}, \mu_{14}, \mu_{15}\}
\]

At last, the number of tuples \( k_3 \) of segment \( s_3 \) is set accordingly:
\[
k_2 := |T_2| = |\{\mu_7, \mu_8, \mu_{11}, \mu_{14}, \mu_{15}\}| = 5
\]

The segment \( s_2 \) after the successful completion of algorithm \( \text{segmentDifferenceInPlace}(s_2, s_3) \) (see algorithm 6) is shown in table 4.18.
Figure 4.10.: Segment $s_2$ after creation and subtraction of segment $s_3$
4.2. Set Operations on Segments

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{\text{select}} )</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>( A_{\text{from}} )</td>
<td>All attributes of relation \textit{post}</td>
</tr>
<tr>
<td>( A_{\text{where}} )</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>( A_{\text{orderby}} )</td>
<td>{id}</td>
</tr>
<tr>
<td>( A_{\text{initial}} )</td>
<td>{created, likes}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_2 )</td>
<td>{post}</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( (\text{created} \geq \text{today}) \land (\text{created} &lt; \text{last_hour}) \land (\text{likes} \geq 20) )</td>
</tr>
<tr>
<td>( O_2 )</td>
<td>((\text{id}, \uparrow)))</td>
</tr>
<tr>
<td>( k_2 )</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{i}_2 )</td>
<td>( (\text{created} \geq \text{last_week}) \land ((\text{likes} &gt; 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))) )</td>
</tr>
<tr>
<td>( \lambda_2 )</td>
<td>( \mu_{16}[\text{id, likes, created}] = (16, 140, \text{last_hour}) ) (see table 4.10)</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>false</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>{( \mu_7, \mu_8, \mu_11, \mu_{14}, \mu_{15} }}</td>
</tr>
</tbody>
</table>

Table 4.18.: Values of segment \( s_2 \) after subtraction of segment \( s_3 \)
4. Segments

4.2.3. Intersection of Two Segments

Similar to the calculation of the difference between two segments, because all segments of a top-k semantic cache are disjoint, we only ever need to calculate the intersection between an existing segment $s_c$ and the newly created segment $s_{new}$ of a query $q_{new}$ (see figure 4.11).

![Figure 4.11: Intersection between two segments of the top-k semantic cache](image)

The intersection of two segments can be calculated in-place in a similar fashion to algorithm segmentDifferenceInPlace($s_c$, $s_{new}$) (see algorithm 6), but it usually is not done this way. Instead, a new segment $s_i$ is created for the intersection of $s_c$ and $s_{new}$ by algorithm segmentIntersection($s_c$, $s_{new}$) (see algorithm 7), because we cannot do the intersection in-place if we have already done the calculation of the difference that way. If we want a segment for the intersection and another segment for the difference, because we use the approach of no coalescing, we will need to create a new segment for either the intersection or the difference.

Analogous to the calculation of the difference, algorithm segmentIntersection($s_c$, $s_{new}$) also requires that all attributes of the new segment are contained in segment $s_c$, i.e., $A_{new}$ ⊆ $A_{select}$.

The algorithm uses segment $s_c$ as kind of a base segment for the intersection. It copies most attribute sets, only updates $A_{where}$ to additionally contain the attributes of the where clause of $s_{new}$. It copies the set of tables $R_c$, the order by $O_c$, the condition $\hat{i}_c$ and the last loaded tuple $\lambda_c$.

Please note that it is entirely possible that the tuple $\lambda_c$ itself is not contained in $s_i$. But the condition $\hat{O}_c \leq \lambda_i$ will still accurately describe the tuples that have been loaded from the server, because $\lambda_c$ will either be the last loaded tuple of $s_i$ or a tuple between the last loaded tuple of $s_i$ (that is actually contained in $s_i$) and the tuple that is transferred from the server next.
4.2. Set Operations on Segments

Algorithm 7 \( s_c \cap s_{\text{new}} \): segmentIntersection\((s_c, s_{\text{new}})\)

Input:
- \( s_c, s_{\text{new}} \)
  - with \( A_{\text{new}}^{\text{where}} \subseteq A_c^{\text{select}} \)

Output:
- \( s_i := s_c \cap s_{\text{new}} \)

1. \( A_i^{\text{select}} := A_c^{\text{select}} \)
2. \( A_i^{\text{from}} := A_c^{\text{from}} \)
3. \( A_i^{\text{where}} := A_c^{\text{where}} \cup A_{\text{new}}^{\text{where}} \)
4. \( A_i^{\text{orderby}} := A_c^{\text{orderby}} \)
5. \( A_i^{\text{initial}} := A_c^{\text{initial}} \)
6. \( R_i := R_c \)
7. \( s_i := s_c \cap s_{\text{new}} \)
8. \( Q_i := Q_c \)
9. \( e_i := e_c \)
10. \( \lambda_i := \lambda_c \)
11. \( c_i := c_c \)
12. \( T_i := \sigma_{s_{\text{new}}}(T_c) \)
13. \( k_i := |T_i| \)
14. \( \text{return } s_i \)

The value of \( c_c \) can also be copied, because the implication \( c_c \rightarrow c_i \) always holds. On the one hand, if segment \( s_c \) has been completely loaded, the intersection of \( s_c \) and \( s_{\text{new}} \) has been completely loaded as well. On the other hand, if segment \( s_c \) has not been completely loaded, the intersection of \( s_c \) and \( s_{\text{new}} \) may or may not have been completely loaded. Therefore, as a conservative approach, we can copy the flag from \( s_c \).

The algorithm moves all tuples that satisfy \( s_{\text{new}} \) and \( s_c \) from \( T_c \) to \( T_i \), i.e., \( T_i := \sigma_{s_{\text{new}}}(T_c) \). Since all tuples that are contained in \( T_i \) satisfy the condition \( s_c \), we do not need to check for it. Finally, \( k_i \) is updated to reflect the new number of tuples of segment \( s_i \).

Example 21 (Intersection of segment \( s_1 \) and segment \( s_2 \)) In this example, we use segment \( s_1 \) before subtraction of segment \( s_2 \) (see table 4.4) and the newly created, empty segment \( s_2 \) of query \( q_2 \) (see table 4.6). We consider the intersection of these two segments. The algorithm segmentIntersection\((s_c, s_{\text{new}})\) (see algorithm 7) creates a new segment \( s_i \).

The algorithm uses most of the attribute sets of segment \( s_1 \) for the segment \( s_i \) of the intersection:

\[
\begin{align*}
    A_i^{\text{select}} &:= A_1^{\text{select}} = \{\text{id}, \text{likes, created, title, author}\} \\
    A_i^{\text{from}} &:= A_1^{\text{from}} \quad \text{All attributes of relation } \text{post} \\
    A_i^{\text{orderby}} &:= A_1^{\text{orderby}} = \{\text{likes, id}\} \\
    A_i^{\text{initial}} &:= A_1^{\text{initial}} = \emptyset
\end{align*}
\]
4. Segments

Only the attribute set $A_i^{\text{where}}$ is set to the union of the corresponding attribute sets of segment $s_1$ and segment $s_2$ to include all attributes that are used by the segment condition $\hat{s}_i$:

$$A_i^{\text{where}} := A_1^{\text{where}} \cup A_2^{\text{where}} = \{\text{created, likes}\} \cup \{\text{created, likes}\} = \{\text{created, likes}\}$$

The segment condition $\hat{s}_i$ is the conjunction of the segment conditions of segment $s_1$ and segment $s_2$:

$$\hat{s}_i := \hat{s}_1 \land \hat{s}_2 = ((\text{created} \geq \text{last week}) \land (\text{likes} \geq 100))$$

$$\land ((\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last hour}) \lor (\text{likes} \geq 20)))$$

$$\equiv (\text{created} \geq \text{today}) \land (\text{likes} \geq 100)$$

All tuples that satisfy the segment condition $\hat{s}_i$ of the new segment $s_i$ are copied into segment $s_i$. Please note that the query $\sigma_{\hat{s}_2}(T_1)$ only needs to apply the condition $\hat{s}_2$, because all tuples in segment $s_1$ naturally satisfy the condition $\hat{s}_1$:

$$T_i := \sigma_{\hat{s}_2}(T_1) = \{\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}\}$$

Table 4.19 shows the result of query $\sigma_{\hat{s}_2}(T_1)$.

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>...</th>
<th>q_1</th>
<th>q_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>220</td>
<td>today</td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>180</td>
<td>today</td>
<td></td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>170</td>
<td>last hour</td>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>230</td>
<td>last hour</td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>190</td>
<td>today</td>
<td></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>200</td>
<td>last hour</td>
<td></td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 4.19.: Result of query $\sigma_{\hat{s}_2}(T_1)$

The number of tuples of segment $s_i$ is set accordingly:

$$k_i := |T_i| = |\{\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}\}| = 6$$

The values of the attributes $R_i$, $O_i$, $\hat{i}_i$, $\lambda_i$ and $c_i$ are also copied from their respective correspondents in segment $s_1$. This means that $\lambda_i$ is set to $\mu_2$, which is a tuple that is not contained in segment $s_i$. But as discussed above, the condition $\hat{o}_i^{\leq \lambda}$ (see definition 10) will still accurately describe the tuples that have been loaded from the server in segment $s_i$.

An overview of all values of segment $s_i$ is given by table 4.20.
4.2. Set Operations on Segments

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_i^{\text{select}}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_i^{\text{from}}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_i^{\text{where}}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_i^{\text{orderby}}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_i^{\text{initial}}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i$</td>
<td>{post}</td>
</tr>
<tr>
<td>$s_i$</td>
<td>(created $\geq$ today) $\land$ (likes $\geq$ 100)</td>
</tr>
<tr>
<td>$O_i$</td>
<td>((likes, $\downarrow$), (id, $\uparrow$))</td>
</tr>
<tr>
<td>$k_i$</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{i}_i$</td>
<td>false</td>
</tr>
<tr>
<td>$\lambda_i$</td>
<td>$\mu_2$[id, likes, created] = (2, 160, last_week) (see table 4.3)</td>
</tr>
<tr>
<td>$c_i$</td>
<td>false</td>
</tr>
<tr>
<td>$T_i$</td>
<td>${\mu_7, \mu_8, \mu_9, \mu_{10}, \mu_{14}, \mu_{19}}$ (see table 4.19)</td>
</tr>
</tbody>
</table>

Table 4.20.: Values of segment $s_i$
4. Segments

4.2.4. Union of Two Segments

In contrast to difference and intersection, the union of two segments can and will target two segments $s_1$ and $s_2$ that are already in the cache. That means that we have to consider that both segments have initially loaded tuples (i.e., $\hat{t}_1$ and $\hat{t}_2$) that we have to account for (see figure 4.12).

![Figure 4.12.: Union of two segments of the top-k semantic cache](image)

We would prefer if the union of two segments could be calculated in-place, because, afterward, we will have two segments that we will not need any more. Therefore, it would be best if we were able to use one of these segments to store the result of the union. In fact, in-place calculation of the union of two segments is possible as shown in algorithms segmentUnionInPlace($s_1$, $s_2$) (see algorithm 8) and segmentUnionInPlace*($s_1$, $s_2$) (see algorithm 9). The first algorithm has a broader application, but the second algorithm creates a better resulting segment with a much shorter and easier condition $\hat{t}$. That is why the top-k semantic cache should opt for algorithm 9 if it is applicable.

4.2.4.1. The General Case

The requirements for algorithm segmentUnionInPlace($s_1$, $s_2$) (see algorithm 8) are as follows:

1. $R_1 \cap R_2 \neq \emptyset$

2. $A_{\text{where}} \cup A_{\text{where}}^{\text{initial}} \cup A_{\text{orderby}}^{\text{initial}} \subseteq A_{\text{from}}^{\text{initial}} \cap A_{\text{from}}^{\text{orderby}}$
Algorithm 8 \( s_1 \cup s_2 \): segmentUnionInPlace\((s_1, s_2)\)

**Input:** \( s_1, s_2 \)
- with \( R_1 \cap R_2 \neq \emptyset \)
- and \( A_1^{\text{where}} \cup A_2^{\text{where}} \cup A_1^{\text{initial}} \cup A_2^{\text{initial}} \cup A_1^{\text{orderby}} \cup A_2^{\text{orderby}} \subseteq A_1^{\text{from}} \cap A_2^{\text{from}} \)

**Output:** \( s_1 := s_1 \cup s_2 \)

1: \( R_1 := R_1 \cap R_2 \)
2: \( A_1^{\text{from}} := A_1^{\text{from}} \cap A_2^{\text{from}} \)
3: \( A_1^{\text{select}} := A_1^{\text{select}} \cap A_2^{\text{select}} \)
4: \( \hat{s}_1 := ((\hat{r}_1 \vee \hat{o}_1^{\leq \lambda}) \wedge \hat{s}_1) \lor ((\hat{r}_2 \vee \hat{o}_2^{\leq \lambda}) \wedge \hat{s}_2) \)
5: \( A_1^{\text{initial}} := A_1^{\text{initial}} \cup A_2^{\text{initial}} \cup A_1^{\text{where}} \cup A_2^{\text{where}} \cup A_1^{\text{orderby}} \cup A_2^{\text{orderby}} \)
6: \( \lambda_1 := \perp \)
7: \( \hat{s}_1 := \hat{s}_1 \lor \hat{s}_2 \)
8: \( A_1^{\text{where}} := A_1^{\text{where}} \cup A_2^{\text{where}} \)
9: \( T_1 := \pi_{A_1^{\text{select}}}(T_1) \cup \pi_{A_2^{\text{select}}}(T_2) \)
10: \( k_1 := k_1 + k_2 \)
11: \( c_1 := c_1 \land c_2 \)
12: \( O_1 := \text{retainAttributes}(O_1, A_1^{\text{select}}) \)
13: \( A_1^{\text{orderby}} := A_1^{\text{orderby}} \cap A_1^{\text{select}} \)
14: \textbf{return} \( s_1 \)
4. Segments

The set of tables of the segment \( s_1 \cup s_2 \) is the intersection of the set of tables of \( s_1 \) and set of the tables of \( s_2 \), i.e., \( R_1 \cap R_2 \). The reason is that, on the one hand, the tables \( R_1 \setminus R_2 \) cannot be part of the segment \( s_1 \cup s_2 \), because the tuples stored in \( s_2 \) do not contain the values for the attributes of \( R_1 \setminus R_2 \). On the other hand, the tuples of \( s_1 \) do not contain the values for the attributes of \( R_2 \setminus R_1 \). In conclusion, the tables of segment \( s_1 \cup s_2 \) must be \((R_1 \cup R_2) \setminus (R_2 \setminus R_1) = R_1 \cap R_2 \). If the intersection of \( R_1 \) and \( R_2 \) is empty, the union of \( s_1 \) and \( s_2 \) will also be empty.

The primary attributes of \( R_1 \cap R_2 \) are contained in \( A_1^{select} \), because they are a subset of the primary attributes of \( A_1^{from} \), which, by definition of a segment (see definition 9), must be contained in \( A_1^{select} \). Analogously, the primary attributes of \( R_1 \cap R_2 \) are contained in \( A_2^{select} \). Hence, they are also contained in the intersection \( A_1^{select} \cap A_2^{select} \). Therefore, \( A_1^{select} \cap A_2^{select} \) cannot be the empty set, and the union of \( s_1 \) and \( s_2 \) is not empty. The first requirement of algorithm 8, thus, ensures that the union of \( s_1 \) and \( s_2 \) is not empty.

The second requirement ensures that all attributes that are used in the conditions of \( s_1 \cup s_2 \) are from the tables \( R_1 \cap R_2 \), because these will be the tables of the segment \( s_1 \cup s_2 \).

The algorithm starts off by updating the tables of the segment \( s_1 \), which will store the union of \( s_1 \) and \( s_2 \), to \( R_1 \cap R_2 \), which will be the tables of \( s_1 \cup s_2 \), as discussed above. \( A_1^{from} \) and \( A_1^{select} \) are updated accordingly.

All tuples of \( s_1 \) and \( s_2 \) will be considered initially inserted tuples of \( s_1 \cup s_2 \) by the algorithm. Therefore, \( \hat{i}_1 \) is assigned the disjunction of the conditions that describe the loaded tuples of \( s_1 \) and \( s_2 \) (see proposition 10). \( A_1^{initial} \) is updated to contain all attributes used in the new \( \hat{i}_1 \), and \( \lambda \) is set to \( \perp \). Then, the where clause of segment \( s_1 \) is set to \( \hat{s}_1 \cup \hat{s}_2 \). And \( A_1^{where} \) is updated.

The algorithm moves all tuples from \( s_2 \) into \( s_1 \). \( k_1 \) is updated to reflect the new number of tuples of segment \( s_1 \).

Please note that information will be lost if \( A_1^{select} \neq A_2^{select} \). Some information loss could be avoided if \( A_1^{select} \) was set to \((A_1^{select} \cup A_2^{select}) \cap (A_1^{from} \cap A_2^{from}) \). But using this approach, the tuples of \( s_1 \) would miss the values for attributes \( A_2^{select} \setminus A_1^{select} \) and the tuples of \( s_2 \) would miss the values for attributes \( A_1^{select} \setminus A_2^{select} \), respectively. We would need to transfer the missing values of the tuples from the server. The approach of this algorithm, however, does not need to access the server.

The value of \( c_1 \) is set to \( c_1 \wedge c_2 \), because, if both segments have been completely loaded, the union of these segments will have been loaded completely as well. But if one of the segments has not been completely loaded, the union of the segments will also not have been loaded completely.

Finally, the algorithm must change the order by in a way that the new one only contains the attributes of the new \( A_1^{select} \). Otherwise, we would not be able to generate the condition that describes the tuples that have been loaded for the segment (see definition 9 and proposition 10).
If the order by clause only consists of attribute orderings, this can easily be achieved by just removing all attribute orderings, whose attributes are not contained in the new $A_{1}^{select}$.

**Example 22 (Union of segment $s_1$ and segment $s_2$)** In this example, we calculate the union of the latest versions of segment $s_1$ (i.e., after subtraction of segment $s_2$, see table 4.12) and segment $s_2$ (i.e., after subtraction of segment $s_3$, see table 4.18).

For easier understanding of the example, we will call the segment that contains the union of segment $s_1$ and segment $s_2$ the segment $s_u$, even though, the algorithm segment-UnionInPlace($s_1$, $s_2$) (see algorithm 8) obviously operates in-place and the union of segment $s_1$ and segment $s_2$ is stored in either segment $s_1$ or segment $s_2$.

The algorithm computes the attribute sets of segment $s_u$ as follows:

\[
A_u^{select} := A_1^{select} \cap A_2^{select} = \{\text{id, likes, created, title, author}\}
\]

\[
A_u^{from} := A_1^{from} \cap A_2^{from} = \text{All attributes of relation post}
\]

\[
A_u^{where} := A_1^{where} \cup A_2^{where} = \{\text{created, likes}\} \cup \{\text{created, likes}\} = \{\text{created, likes}\}
\]

\[
A_u^{orderby} := A_1^{orderby} \cap A_2^{orderby} = \{\text{likes, id}\}
\]

\[
A_u^{initial} := A_1^{initial} \cup A_2^{initial} \cup A_1^{where} \cup A_2^{where} \cup A_1^{orderby} \cup A_2^{orderby} \\
= \emptyset \cup \{\text{created, likes}\} \cup \{\text{created, likes}\} \cup \{\text{created, likes}\} \\
\quad \cup \{\text{likes, id}\} \cup \{\text{id}\} \\
= \{\text{created, likes, id}\}
\]

Trivially, the set of relations $R_u$ of segment $s_u$ is

\[
R_u := R_1 \cap R_2 = \{\text{post}\} \cap \{\text{post}\} = \{\text{post}\}.
\]

The order by $O_u$ of segment $s_u$ can, for example, be set to the order by $O_1$ of segment $s_1$:

\[
O_u := \text{retainAttributes}(O_1, A_1^{select} \cap A_2^{select}) = ((\text{likes}, \downarrow), (\text{id}, \uparrow))
\]

The last loaded tuple $\lambda_u$ is set to $\perp$. And the completion flag $c_u$ is set to $c_u := c_1 \land c_2 = \text{false} \land \text{false} \equiv \text{false}$.

The expression $\hat{s}_1$ is known from table 4.12:

\[
\hat{s}_1 = (\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)
\]

And the expression $\hat{s}_2$ is contained in table 4.18:

\[
\hat{s}_2 = (\text{created} \geq \text{today}) \land (\text{created} < \text{last hour}) \land (\text{likes} \geq 20)
\]

The algorithm uses the expressions $\hat{s}_1$ and $\hat{s}_2$ to determine $\hat{s}_u$:

\[
\hat{s}_u := \hat{s}_1 \lor \hat{s}_2 \\
= ((\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)) \lor ((\text{created} \geq \text{today}) \land (\text{created} < \text{last hour}) \land (\text{likes} \geq 20))
\]

100
4. Segments

The expression $\hat{i}_1$ can be taken from table 4.12. And the expression $\hat{o}_1^{\leq \lambda}$ has been calculated in example 13:

$$\hat{i}_1 = \text{false}$$

$$\hat{o}_1^{\leq \lambda} = (\text{likes is } \bot) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))$$

Similarly, the expression $\hat{i}_2$ can be taken from table 4.18. And the expression $\hat{o}_2^{\leq \lambda}$ has been calculated in example 19:

$$\hat{i}_2 = (\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$$

$$\hat{o}_2^{\leq \lambda} = \text{id} \leq 16$$

Using these expressions, the algorithm calculates the expression $\hat{i}_u$ as follows:

$$\hat{i}_u := (((\hat{i}_1 \lor \hat{o}_1^{\leq \lambda}) \land \hat{s}_1) \lor ((\hat{i}_2 \lor \hat{o}_2^{\leq \lambda}) \land \hat{s}_2))$$

$$= (((\text{false}) \lor ((\text{likes is } \bot) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$

$$\land (((\text{created} \geq \text{last\_week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)))$$

$$\lor (((\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$

$$\land (\text{id} \leq 16))$$

$$\lor (((\text{created} \geq \text{today}) \land (\text{created} \land \text{last\_hour}) \land (\text{likes} \geq 20)))$$

$$\land (((\text{created} \geq \text{last\_week}) \land (\text{created} \land \text{today})))$$

$$\lor (((\text{created} \geq \text{last\_week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$

$$\land (\text{id} \leq 16))$$

$$\lor (((\text{created} \geq \text{today}) \land (\text{created} \land \text{last\_hour}) \land (\text{likes} \geq 20)))$$

$$\equiv (((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$

$$\land (\text{created} \geq \text{last\_week})$$

$$\lor (((\text{created} \geq \text{today}) \land (\text{created} \land \text{last\_hour}) \land (\text{likes} \geq 20)))$$

$$\equiv (((\text{created} \geq \text{last\_week}) \land (\text{created} \land \text{last\_hour}))$$

$$\lor (((\text{created} \geq \text{today}) \land (\text{created} \land \text{last\_hour}) \land (\text{likes} \geq 20)))$$
Finally, the algorithm merges the tuple sets $T_1$ and $T_2$ from segment $s_1$ and segment $s_2$:

$$T_u := \pi_{A_1^{select} \cap A_2^{select}}(T_1) \cup \pi_{A_1^{select} \cap A_2^{select}}(T_2)$$

$$= \{\mu_1, \mu_2, \mu_3, \mu_4\} \cup \{\mu_7, \mu_8, \mu_{11}, \mu_{14}, \mu_{15}\}$$

$$= \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_7, \mu_8, \mu_{11}, \mu_{14}, \mu_{15}\}$$

And accordingly, the number of tuples of segment $s_u$ is set:

$$k_u := k_1 + k_2 = 4 + 5 = 9$$

Table 4.22 provides an overview of all values of the segment $s_u := s_1 \cup s_2$.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_u^{select}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_u^{from}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_u^{where}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_u^{orderby}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_u^{initial}$</td>
<td>{created, likes, id}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_u$</td>
<td>{post}</td>
</tr>
<tr>
<td>$s_u$</td>
<td>((created $\geq$ last_week) $\land$ (created $&lt;$ today) $\land$ (likes $\geq$ 100)) $\lor$ ((created $\geq$ today) $\land$ (created $&lt;$ last_hour) $\land$ (likes $\geq$ 20))</td>
</tr>
<tr>
<td>$O_u$</td>
<td>((likes, ↓), (id, ↑))</td>
</tr>
<tr>
<td>$k_u$</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{r}_u$</td>
<td>((created $\geq$ last_week) $\land$ (created $&lt;$ last_hour) $\land$ ((likes $&gt; 160$) $\lor$ ((likes = 160) $\land$ (id $\leq 2$)))) $\lor$ ((id $\leq 16$) $\land$ (created $\geq$ today) $\land$ (created $&lt;$ last_hour) $\land$ (likes $\geq$ 20)))</td>
</tr>
<tr>
<td>$\lambda_u$</td>
<td>⊥</td>
</tr>
<tr>
<td>$c_u$</td>
<td>false</td>
</tr>
<tr>
<td>$T_u$</td>
<td>{\mu_1, \mu_2, \mu_3, \mu_4, \mu_7, \mu_8, \mu_{11}, \mu_{14}, \mu_{15}}</td>
</tr>
</tbody>
</table>

Table 4.22.: Values of segment $s_u := s_1 \cup s_2$

### 4.2.4.2. A Special Case

Algorithm segmentUnionInPlace*($s_1, s_2$) (see algorithm 9) has much stricter requirements than algorithm segmentUnionInPlace*(s1, s2) (see algorithm 8). It can only calculate the union of two segments that have the same tables and use the same ordering, i.e., $R_1 = R_2$ and $O_1 = O_2$. 
4. Segments

Algorithm 9 \( s_1 \cup^* s_2 \): segmentUnionInPlace\(^*\)(\(s_1, s_2\))

\[\begin{align*}
\text{Input:} & \quad s_1, s_2 \\
& \quad \text{with } R_1 = R_2, \ O_1 = O_2, \\
& \quad \text{and } \lambda_1[A_1^{\text{select}} \cap A_2^{\text{select}}] < O_1 \lambda_2[A_1^{\text{select}} \cap A_2^{\text{select}}] \\
\text{Output:} & \quad s_1 := s_1 \cup^* s_2 \\
1: & \quad \hat{i}_1 := \hat{i}_1 \lor ((\hat{i}_2 \lor \hat{o}_2) \land \hat{s}_2) \\
2: & \quad A_1^{\text{initial}} := A_1^{\text{initial}} \cup A_2^{\text{initial}} \cup A_2^{\text{where}} \cup A_2^{\text{orderby}} \\
3: & \quad A_1^{\text{select}} := A_1^{\text{select}} \cap A_2^{\text{select}} \\
4: & \quad \hat{s}_1 := \hat{s}_1 \lor \hat{s}_2 \\
5: & \quad A_1^{\text{where}} := A_1^{\text{where}} \cup A_2^{\text{where}} \\
6: & \quad T_1 := \pi_{A_1^{\text{select}}}(T_1) \cup \pi_{A_1^{\text{select}}}(T_2) \\
7: & \quad \lambda_1 := \lambda_1[A_1^{\text{select}}] \\
8: & \quad k_1 := k_1 + k_2 \\
9: & \quad c_1 := c_1 \land c_2 \\
10: & \quad \text{return } s_1
\end{align*}\]

By definition of a segment (see definition 9), we know that \( A_1^{\text{orderby}} \subseteq A_1^{\text{select}} \) and \( A_2^{\text{orderby}} \subseteq A_2^{\text{select}} \). And if the orderings are identical, the attributes used in those orderings are identical as well, i.e., \( A_1^{\text{orderby}} = A_2^{\text{orderby}} \). Therefore, we can conclude that \( A_1^{\text{orderby}} = A_2^{\text{orderby}} \subseteq A_1^{\text{select}} \cap A_2^{\text{select}} \). That is why, in contrast to the previous algorithm, this algorithm is able to preserve the ordering.

The tuples \( \lambda_1[A_1^{\text{select}} \cap A_2^{\text{select}}] \) and \( \lambda_2[A_1^{\text{select}} \cap A_2^{\text{select}}] \) are ordered with respect to \( O_1 \), because \( O_1 \) is a total ordering by definition of a segment (see definition 9). Without loss of generality, we assume that \( \lambda_1[A_1^{\text{select}} \cap A_2^{\text{select}}] < O_1 \lambda_2[A_1^{\text{select}} \cap A_2^{\text{select}}] \).

Then, the tuples of \( s_2 \) must be treated as initially loaded tuples. Hence, \( \hat{i}_1 \) is assigned the disjunction of \( \hat{i}_1 \) and an expression that describes all tuples that have been loaded for \( s_2 \) (see proposition 10). \( A_1^{\text{initial}} \) is updated to contain all attributes of the new \( \hat{i}_1 \).

The new select clause is set to \( A_1^{\text{select}} \cap A_2^{\text{select}} \), because all tuples of \( T_1 \) miss the values for the attributes \( A_2^{\text{select}} \setminus A_1^{\text{select}} \), and, analogously, the tuples of \( T_2 \) miss the values for the attributes \( A_1^{\text{select}} \setminus A_2^{\text{select}} \).

The where clause of segment \( s_1 \) is set to \( \hat{s}_1 \lor \hat{s}_2 \), and \( A_1^{\text{where}} \) is updated accordingly.

The last loaded tuples \( \lambda_1 \) can stay essentially the same, because the tuples that have been moved into \( s_1 \) from \( s_2 \) are considered initially inserted tuples and thus have no influence on \( \lambda_1 \). Only the attributes that are no longer used by the segment will have to be removed.

During the calculation of the union, attributes may be removed, but no tuple will be deleted. Therefore, \( k_1 \) is set to \( k_1 + k_2 \). The value of \( c_1 \) is set to \( c_1 \land c_2 \) with the same reasoning as for algorithm 8.
4.2. Set Operations on Segments

Example 23 (Union* of segment \(s_1\) and segment \(s_3\)) In this example, we calculate the union of the latest versions of segment \(s_1\) (i.e., after subtraction of segment \(s_2\), see table 4.12) and segment \(s_3\) (i.e., after execution of its remainder query \(r_3\), see table 4.17). For easier understanding of the example, similar to the last example, we will call the segment that contains the union of segment \(s_1\) and segment \(s_3\) the segment \(s_u^*\), even though, the algorithm \(\text{segmentUnionInPlace}^*(s_1, s_3)\) (see algorithm 9) clearly operates in-place.

The requirements of the algorithm are met, because \(R_1 = \{\text{post}\} = R_3\) and \(O_1 = ((\text{likes}, \downarrow), (\text{id}, \uparrow)) = O_3\) hold.

The last loaded tuple \(\lambda_1\) of segment \(s_1\) is
\[
\lambda_1 = \mu_2 \text{ with } \mu_2[\text{id}, \text{likes}, \text{created}] = (2, 160, \text{last}\_\text{week}).
\]

On the other hand, the last loaded tuple \(\lambda_3\) of segment \(s_3\) is
\[
\lambda_3 = \mu_{27} \text{ with } \mu_{27}[\text{id}, \text{likes}, \text{created}] = (27, 50, \text{last}\_\text{hour}).
\]

Because the inequality \(\lambda_1 < O_1, \lambda_3\) holds, the segment \(s_1\) must be the base segment of the union of the segment \(s_1\) and segment \(s_3\).

Hence, the attribute sets of segment \(s_u^*\) are set as follows:

\[
\begin{align*}
A_{u_\text{select}}^{\text{select}} &= A_1^{\text{select}} \cap A_3^{\text{select}} = \{\text{id}, \text{likes}, \text{created}, \text{title}, \text{author}\} \\
A_{u_\text{where}}^{\text{where}} &= A_1^{\text{where}} \cup A_3^{\text{where}} = \{\text{created}, \text{likes}\} \cup \{\text{created}\} = \{\text{created}, \text{likes}\} \\
A_{u_\text{initial}}^{\text{initial}} &= A_1^{\text{initial}} \cup A_3^{\text{initial}} \cup A_3^{\text{where}} \cup A_3^{\text{orderby}} \cup \emptyset \cup \{\text{created}, \text{likes}, \text{id}\} \cup \{\text{created}\} \cup \{\text{likes}, \text{id}\} \\
&= \{\text{created}, \text{likes}, \text{id}\}
\end{align*}
\]

Table 4.12 contains the expression \(\hat{s}_1\):

\[
\hat{s}_1 = (\text{created} \geq \text{last}\_\text{week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)
\]

And table 4.17 contains the expression \(\hat{s}_3\):

\[
\hat{s}_3 = \text{created} \geq \text{last}\_\text{hour}
\]

Combining the expressions \(\hat{s}_1\) and \(\hat{s}_3\), the algorithm calculates the following expression \(\hat{s}_{u^*}\):

\[
\begin{align*}
\hat{s}_{u^*} &= \hat{s}_1 \lor \hat{s}_3 \\
&= ((\text{created} \geq \text{last}\_\text{week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)) \\
&\lor (\text{created} \geq \text{last}\_\text{hour})
\end{align*}
\]
4. Segments

We determine the condition $\hat{o}_3^{\leq \lambda}$ using algorithm $\text{conditionOrderBy}(O_3, \lambda_3)$ (see algorithm 2):

$$
\hat{o}_3^{\leq \lambda} := \text{conditionOrderBy}(O_3, \lambda_3)
= \text{conditionOrderBy}(((\text{likes}, \downarrow), (\text{id}, \uparrow)), (27, 50, \text{last hour}, \ldots))
= (\text{likes is } \bot) \lor (\text{likes } > 50) \lor ((\text{likes } = 50) \land \text{conditionOrderBy}(((\text{id}, \uparrow)), (27, 50, \text{last hour}, \ldots)))
= (\text{likes is } \bot) \lor (\text{likes } > 50) \lor ((\text{likes } = 50) \land (\text{id } \leq 27))
$$

The expression $\hat{i}_1$ and $\hat{i}_3$ can be taken from table 4.12 or table 4.17 respectively:

$$
\hat{i}_1 = \text{false}
\hat{i}_3 = ((\text{created } \geq \text{last hour}) \lor ((\text{created } \geq \text{today}) \land (\text{likes } \geq 20)))
\land ((\text{likes } > 160) \lor (\text{id } \leq 16))
$$

Using these expressions, the algorithm calculates $\hat{i}_{us}$ as follows:

$$
\hat{i}_{us} := \hat{i}_1 \lor ((\hat{i}_3 \lor \hat{o}_3^{\leq \lambda}) \land \hat{s}_3)
= (\text{false})
\lor (((((\text{created } \geq \text{last hour}) \lor ((\text{created } \geq \text{today}) \land (\text{likes } \geq 20)))
\land ((\text{likes } > 160) \lor (\text{id } \leq 16)))
\lor ((\text{likes is } \bot) \lor (\text{likes } > 50) \lor ((\text{likes } = 50) \land (\text{id } \leq 27))))
\land (\text{created } \geq \text{last hour}))
= (((\text{created } \geq \text{last hour}) \lor ((\text{created } \geq \text{today}) \land (\text{likes } \geq 20)))
\land ((\text{likes } > 160) \lor (\text{id } \leq 16)))
\lor ((\text{likes is } \bot) \lor (\text{likes } > 50) \lor ((\text{likes } = 50) \land (\text{id } \leq 27))))
\land (\text{created } \geq \text{last hour})
= (((\text{created } \geq \text{last hour}) \lor ((\text{created } \geq \text{today}) \land (\text{likes } \geq 20)))
\land ((\text{likes } > 160) \lor (\text{id } \leq 16)))
\lor ((\text{likes is } \bot) \lor (\text{likes } > 50) \lor ((\text{likes } = 50) \land (\text{id } \leq 27))))
\land (\text{created } \geq \text{last hour}))
$$

Finally, the algorithm merges the tuple sets $T_1$ and $T_3$ from segment $s_1$ and segment $s_3$:

$$
T_{us} := \pi_{A_1^{\text{select}}} \cap \pi_{A_3^{\text{select}}}(T_1) \cup \pi_{A_1^{\text{select}}} \cap \pi_{A_3^{\text{select}}}(T_3)
= \{\mu_1, \mu_2, \mu_3, \mu_4\} \cup \{\mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\}
= \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\}
$$
4.2. Set Operations on Segments

In addition, the algorithm sets the number of tuples $k_{u_s}$ of segment $s_{u_s}$:

$$k_{u_s} := k_1 + k_2 = 4 + 11 = 15$$

An overview of all values of the segment $s_{u_s} := s_1 \cup^* s_3$ is provided by table 4.24.

<table>
<thead>
<tr>
<th>Attribute sets</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{select}$</td>
<td>{id, likes, created, title, author}</td>
</tr>
<tr>
<td>$A_{from}$</td>
<td>All attributes of relation post</td>
</tr>
<tr>
<td>$A_{where}$</td>
<td>{created, likes}</td>
</tr>
<tr>
<td>$A_{orderby}$</td>
<td>{likes, id}</td>
</tr>
<tr>
<td>$A_{initial}$</td>
<td>{created, likes, id}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query parts</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{u_s}$</td>
<td>{post}</td>
</tr>
<tr>
<td>$\hat{s}_{u_s}$</td>
<td>(created (\geq) last_week) (\land) (created (&lt;) today) (\land) (likes (\geq) 100)) (\lor) (created (\geq) last_hour)</td>
</tr>
<tr>
<td>$O_{u_s}$</td>
<td>(likes, ↓), (id, ↑)</td>
</tr>
<tr>
<td>$k_{u_s}$</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{i}_{u_s}$</td>
<td>(((created (\geq) last_hour) (\lor) (((created (\geq) today) (\land) (likes (\geq) 20))) (\land) (((likes (&gt;) 160) (\lor) (id (\leq) 16))) (\lor) (((likes is (\bot)) (\lor) (likes (&gt;) 50)) (\lor) (((likes = 50) (\land) (id (\leq) 27))) (\land) (created (\geq) last_hour))))</td>
</tr>
<tr>
<td>$\lambda_{u_s}$</td>
<td>$\mu_2[\text{id, likes, created}] = (2, 160, \text{last_week})$</td>
</tr>
<tr>
<td>$c_{u_s}$</td>
<td>false</td>
</tr>
<tr>
<td>$T_{u_s}$</td>
<td>{\mu_1, \mu_2, \mu_3, \mu_4, \mu_9, \mu_{10}, \mu_{12}, \mu_{13}, \mu_{16}, \mu_{19}, \mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}}</td>
</tr>
</tbody>
</table>

Table 4.24.: Values of segment $s_{u_s} := s_1 \cup^* s_3$
5. Pipelining

This chapter proposes an algorithm for the estimation of query bounds that operates on multidimensional histograms. With this algorithm, the top-k semantic cache is able to pipeline query results of queries that can only be partially answered by the cache.

5.1. Multidimensional Histograms

Multidimensional histograms can consist of a combination of a multidimensional structure like grid, phased \[MD88\], or mhist-p \[PI97, LI11\] and a one-dimensional partition algorithm like equi-width \[Koo80, PSC84\], equi-depth \[PSC84\], V-optimal \[JKM98, PGI99, Guh05\], or Maxdiff \[PHIS96, AE04, GTD05\]. Sven Rausch did a survey of these different multidimensional structures and one-dimensional partition algorithms \[Rau14\]. Furthermore, there exist multidimensional histograms with more complex structures like STHoles \[BCG01\].

Please note that the generation of an optimal multidimensional equi-depth histogram is an NP-hard problem \[MPS99\]. But it can be heuristically generated by a single SQL query using \textit{window functions} (see appendix A). IQCache, the prototype of a top-k semantic cache, uses this heuristic approach. Window functions are available in SQL since SQL:2003 \[ISO03\].

Traditionally, in database systems, histograms are used in the query optimizer during creation of the query execution plan to estimate result sizes of (sub-)queries \[Ioa03\]. Most modern database systems use one-dimensional histograms (see table 5.1). They work under the \textit{attribute value independence assumption} \[Ioa03\], even though, it is long known that this can have a very negative impact on query execution performance \[PI97\].

<table>
<thead>
<tr>
<th>Product</th>
<th>Version</th>
<th>Producer</th>
<th>Histogram type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>12c</td>
<td>Oracle</td>
<td>Equi-width, equi-depth [Ora14]</td>
</tr>
<tr>
<td>DB2</td>
<td>11</td>
<td>IBM</td>
<td>Equi-depth [BBB14]</td>
</tr>
<tr>
<td>MariaDB</td>
<td>10.0.2</td>
<td>Monty Program AB</td>
<td>Equi-depth [Mar15]</td>
</tr>
<tr>
<td>MSSQL</td>
<td>2008</td>
<td>Microsoft</td>
<td>Equi-depth [HA09]</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.4</td>
<td>PostgreSQL</td>
<td>Equi-depth [Pos15]</td>
</tr>
</tbody>
</table>

Table 5.1.: Employed histogram types in commercial and free database systems

In addition, histograms can be used to find semantic errors in SQL queries \[PI97\]. And
histograms, especially multidimensional ones, can be employed to estimate query results of long running queries, in particular aggregate queries [PG199, IP99, WS03].

In this chapter, we introduce a new application of multidimensional histograms. We propose an algorithm (see algorithm 14) that uses multidimensional histograms to estimate lower bounds of query results.

5.1.1. Definition

We define multidimensional histograms as follows:

**Definition 19 (Multidimensional histogram)** A multidimensional histogram is described by the following tuple:

\[ H = (A, R, B, I) \]

The attribute set \( A \) contains the attributes, i.e., the dimensions, of the histogram. \( R \) is the underlying relation. The set \( B \) contains all buckets of the histogram (see definition 20). \( I \) describes the index structure of the histogram.

The most important part of a histogram are, of course, its buckets. We define buckets as follows:

**Definition 20 (Bucket)** A bucket \( b \) of a histogram is described by the following tuple:

\[ b = (l, u, t) \]

The lower bound \( l \) and the upper bound \( u \) define a hyperrectangle, which is the span, i.e., defined area, of the bucket. The function \( t : T \rightarrow \mathbb{N} \) represents the different tuple counts of the bucket.

Traditionally, a bucket has only one tuple count, i.e., one number of tuples that are contained in this bucket. In contrast, our bucket definition allows for more than one tuple count. This allows for a more detailed description if the tuples of the bucket can be stored at different locations, e.g., in a client-server architecture.

**Example 24 (Tuple counts)** For example, assume that a total amount of 42 tuples is contained in a bucket \( b = (l, u, t) \). 28 of them are currently stored in the cache. Using the tuple count function \( t \), we are able to precisely describe this situation by setting \( T := \{ \text{cache, server, total} \} \) and assigning \( t(\text{cache}) := 28 \), \( t(\text{server}) := 14 \), and \( t(\text{total}) := 42 \).
5.2. Histograms for the Estimation of Query Bounds

In the following section 5.3, we introduce a new algorithm that can estimate a lower bound for the result of a given query. The algorithm operates on multidimensional histograms. Therefore, the cache needs to prepare the needed multidimensional histograms. It has to keep them up-to-date. But before, it has to select the right multidimensional histograms to create in the first place. It has to select the multidimensional histograms that best balance the trade-off between storage space and query coverage as well as accuracy of the estimated lower bounds.

This section consists of two parts. First, we investigate the properties that a multidimensional histogram must have so that it can be used to estimate a bound for the result of a given query (see section 5.2.1). Secondly, we show that the optimal selection of multidimensional histograms for a known query load and a given space restriction is indeed NP-complete (see section 5.2.2).

5.2.1. Properties of Histograms for the Estimation of Query Bounds

Based on the definition of a histogram (see definition 19), we define a multidimensional histogram that can be used to estimate a bound for the result of a given query as follows:

**Definition 21 (Histogram for the estimation of query bounds)**

A multidimensional histogram that is usable for the estimation of a bound for the result of a query can be described by the following tuple:

\[ H^e = (A^{grid}, A^{all}, R, B, I) \]

The attribute set \( A^{grid} \) contains all attributes that have buckets that have a grid structure. The attribute set \( A^{all} \) contains all dimensions, i.e., attributes, of the histogram \( H^e \). Hence, the inclusion \( A^{grid} \subseteq A^{all} \) always holds. As in the definition of a histogram (see definition 19), \( R \) is the underlying relation, which can be a join of two or more tables. The set \( B \) contains all buckets of the histogram (see definition 20). And \( I \) describes the index structure of the histogram.

Naturally, not every multidimensional histogram can be used for just any query. To be able to be used to estimate a lower bound of the result of a given query using algorithm 14, a multidimensional histogram must fulfill certain criteria.

**Definition 22 (Usability of a histogram for a specific query)** Let \( H^e = (A^{grid}, A^{all}, R_{H^e}, B, I) \) be a histogram for estimation of query bounds (see definition 21). Let \( q = (\ldots, A_{where}, A_{orderby}, R_q, q, \ldots) \) be a query (see definition 8). The histogram \( H^e \) is usable to...
5. Pipelining

estimate a lower bound of the result of query \( q \) iff.

\[
A_{\text{orderby}} \subseteq A_{\text{grid}} \\
\land \ A_{\text{where}} \subseteq A_{\text{all}} \\
\land \ \Pi_{A_{\text{where}}} (\sigma_{q} (\times_{r \in \mathcal{R}_{q}} r)) \subseteq \Pi_{A_{\text{where}}} (R_{H_{e}}).
\]

First, all attributes that are used in the order by of the query \( q \) must be part of the grid structure of the histogram \( H_{e} \), i.e., \( A_{\text{orderby}} \subseteq A_{\text{grid}} \). This condition ensures that the histogram can be efficiently accessed in correspondence to the order by of query \( q \). Actually, the condition \( A_{\text{orderby}} \subseteq A_{\text{grid}} \) is sufficient, but not necessary. If it holds, the histogram supports any attribute-based lexicographical ordering that can be built by the attributes of the set \( A_{\text{orderby}} \) (see definition 12). But if only certain order bys need to be supported, e.g., if the order by always is an attribute-based lexicographical ordering of the same attributes in the same order, algorithm 14 will not need a grid index at all. It will just need a histogram that is structured based on the order of the attributes in the attribute-based lexicographical ordering.

Secondly, we demand that all attributes that are used in the where condition of query \( q \) must be present in the histogram \( H_{e} \), i.e., \( A_{\text{where}} \subseteq A_{\text{all}} \). This requirement enables several optimizations in algorithm 14 that greatly positively impact its performance. Therefore, while this condition is not necessary to estimate a lower bound in theory, in practice, it is of paramount importance.

The third condition just states that all tuples that can be contained in the result of the query \( q \) must be contained in the underlying relation of the histogram \( H_{e} \).

With the definition of the usability of a multidimensional histogram for a specific query in mind, we can define the following inclusion between two histograms for the estimation of query bounds.

**Definition 23 (Inclusion between histograms)** Let \( H_{1}^{e} = (A_{1}^{\text{grid}}, A_{1}^{\text{all}}, R_{1}, B_{1}, I_{1}) \) and \( H_{2}^{e} = (A_{2}^{\text{grid}}, A_{2}^{\text{all}}, R_{2}, B_{2}, I_{2}) \) be histograms for estimation of query bounds (see definition 21). Then, we define the inclusion between histogram \( H_{1}^{e} \) and histogram \( H_{2}^{e} \) as follows:

\[
H_{1}^{e} \subseteq H_{2}^{e} \iff \ A_{1}^{\text{grid}} \subseteq A_{2}^{\text{grid}} \\
\land \ A_{1}^{\text{all}} \subseteq A_{2}^{\text{all}} \\
\land \ \Pi_{A_{1}^{\text{all}}} (R_{1}) \subseteq \Pi_{A_{2}^{\text{all}}} (R_{2})
\]

The inclusion between histograms as defined in definition 23 is correct, i.e., well-defined, in the sense that inclusion implies usability for estimation of bounds of query results:

**Proposition 16 (Correctness of inclusion between histograms)** Let \( H_{1}^{e} = (A_{1}^{\text{grid}}, A_{1}^{\text{all}}, R_{1}, B_{1}, I_{1}) \) and \( H_{2}^{e} = (A_{2}^{\text{grid}}, A_{2}^{\text{all}}, R_{2}, B_{2}, I_{2}) \) be histograms for estimation of query
5.2. Histograms for the Estimation of Query Bounds

bounds (see definition 21). Let \( q = (\ldots, \text{\texttt{A where}}, \text{\texttt{A order by}}, R_q, \hat{q}, \ldots) \) be a query (see definition 8). Then, the following implication holds:

\[
(H^e_1 \subseteq H^e_2 \text{ and } H^e_1 \text{ usable for } q) \Rightarrow (H^e_2 \text{ usable for } q)
\]

**Proof.** Assume that \( H^e_1 \) is usable for \( q \) (see definition 22), i.e.,

\[
\text{\texttt{A order by}} \subseteq A^\text{grid}_1, \\
\text{\texttt{A where}} \subseteq A^\text{all}_1 \text{ and } \\
\Pi_{A^\text{where} \cup A^\text{order by}}(\sigma_{\hat{q}}(\times_{r \in R_q} r)) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(R_1).
\]

Further, assume that \( H^e_1 \subseteq H^e_2 \) (see definition 23), i.e.,

\[
A^\text{grid}_1 \subseteq A^\text{grid}_2, \\
A^\text{all}_1 \subseteq A^\text{all}_2 \text{ and } \\
\Pi_{A^\text{all}}(R_1) \subseteq \Pi_{A^\text{all}}(R_2).
\]

\( A^\text{order by} \subseteq A^\text{grid}_1 \subseteq A^\text{grid}_2 \) yields

\[
A^\text{order by} \subseteq A^\text{grid}_2 \text{ (*)}.
\]

\( A^\text{where} \subseteq A^\text{all}_1 \subseteq A^\text{all}_2 \) yields

\[
A^\text{where} \subseteq A^\text{all}_2 \text{ (**).}
\]

Because \( A^\text{grid}_1 \subseteq A^\text{all}_1 \) (see definition 21), \( A^\text{order by} \subseteq A^\text{grid}_1 \subseteq A^\text{all}_1 \) and \( A^\text{where} \subseteq A^\text{all}_1 \), it holds that \( A^\text{where} \cup A^\text{order by} \subseteq A^\text{all}_1 \).

It follows that

\[
\Pi_{A^\text{where} \cup A^\text{order by}}(\Pi_{A^\text{all}}(R_1)) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(\Pi_{A^\text{all}}(R_2)),
\]

which simplifies to

\[
\Pi_{A^\text{where} \cup A^\text{order by}}(R_1) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(R_2).
\]

Combining this result with the first assumption,

\[
\Pi_{A^\text{where} \cup A^\text{order by}}(\sigma_{\hat{q}}(\times_{r \in R_q} r)) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(R_1) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(R_2),
\]

yields

\[
\Pi_{A^\text{where} \cup A^\text{order by}}(\sigma_{\hat{q}}(\times_{r \in R_q} r)) \subseteq \Pi_{A^\text{where} \cup A^\text{order by}}(R_2).
\]

Together, the results (*) and (**), and the latter result fulfill definition 22 for \( H^e_2 \) and \( q \), i.e., \( H^e_2 \) is usable for \( q \). \( \square \)
This result allows us to reduce the amount of histograms, the top-k semantic cache needs to store and to maintain. For example, assume that, for a known query load, we need the histograms \( H_1, \ldots, H_n \) \((n \in \mathbb{N})\). But there exists a histogram \( H_0 \), which \( \forall i \in \{1, \ldots, n\}: H_i \subseteq H_0 \) holds for. Then, the top-k semantic cache will only need to create histogram \( H_0 \) to be able to use algorithm \([14]\) which can calculate the lower bounds of query results.

### 5.2.2. NP-Completeness of Optimal Histogram Selection

In the last section, we showed that there is not the one multidimensional histogram that can be used to calculate the lower bound of the result of a certain query with algorithm \([14]\). But instead using the defined and proven inclusion between histograms for estimation of query bounds (see definition \([23]\)), we can select the optimal set of histograms for a known query load under a given memory space restriction. We assume that the benefit of caching a query has been determined for each query based on its execution cost and frequency in the known query load.

**Definition 24 (Histogram selection (HS))** Let \( q_1, \ldots, q_m \) be \( m \) queries with benefits \( b_1, \ldots, b_m \) and \( q_i = (\ldots, A_i^\text{where}, A_i^\text{orderby}, \ldots) \) for all \( i \in \{1, \ldots, m\} \) (see definition \([8]\)). Let \( h_1, \ldots, h_n \) be \( n \) histograms with required memory space \( s_1, \ldots, s_n \) and \( h_j = (A_j^\text{grid}, A_j^\text{all}, \ldots) \) for all \( j \in \{1, \ldots, n\} \) (see definition \([21]\)). And let \( S \) be the amount of available memory space for histogram storage. Then, optimal histogram selection can be expressed by the following optimization problem with side condition:

\[
\begin{align*}
\max_{J \subseteq \{1, \ldots, n\}} & \sum_{i \in \{1, \ldots, m\} \land \exists j \in J: (A_i^\text{orderby} \subseteq A_j^\text{grid} \land A_i^\text{where} \subseteq A_j^\text{all})} b_i \\
\text{s.t.} & \sum_{j \in J} s_j \leq S
\end{align*}
\]

The 0-1 knapsack problem is a widely studied combinatorial optimization problem, which is known to be NP-complete.

**Definition 25 (0-1 knapsack)** \([\text{Sah}75]\) Let there be \( n \) items. Each item has a benefit \( b_i \), a required space \( s_i \) \((i \in \{1, \ldots, n\})\), and can only be chosen at most once. The 0-1 knapsack problem describes the optimization of the total benefit of the chosen items under a space constraint \( S \):

\[
\begin{align*}
\max_{J \subseteq \{1, \ldots, n\}} & \sum_{j \in J} b_j \\
\text{s.t.} & \sum_{j \in J} s_j \leq S
\end{align*}
\]
Using the 0-1 knapsack problem, we can prove that the histogram selection problem is unfortunately an NP-hard problem.

**Proposition 17 (HS is NP-hard)** The histogram selection problem (HS, see definition 24) is NP-hard, because there exists a polynomial-time (actually, linear-time) many-one reduction from the 0-1 knapsack problem (see definition 25) to the histogram selection problem, i.e., 0-1 knapsack \( \leq^P \) HS.

**Proof.** For each item \( j \) (\( j \in \{1, \ldots, n\} \)), create an attribute \( a_j \). Using this attribute, create a query \( q_j \) with benefit \( b_j \) and \( q_j \) := \((\ldots, A_j^{\text{where}}, A_j^{\text{orderby}}, \ldots)\) with \( A_j^{\text{where}} := \{a_j\} \) and \( A_j^{\text{orderby}} := \{a_j\} \) as well. In addition, for each item \( j \) (\( j \in \{1, \ldots, n\} \)), create a histogram \( h_j \) with needed space \( s_j \) and \( h_j := (A_j^{\text{grid}}, A_j^{\text{all}}, \ldots) \) with \( A_j^{\text{grid}} := \{a_j\} \) and \( A_j^{\text{all}} := \{a_j\} \). With these queries and histograms, the histogram selection problem is simplified to the 0-1 knapsack problem:

\[
\begin{align*}
\max & \sum_{j \in \{1, \ldots, n\}} b_j \\
\text{s.t.} & \sum_{j \in J} s_j \leq S
\end{align*}
\]

Equivalence (1) holds because \( A_i^{\text{where}} \subseteq A_j^{\text{all}} \iff \{a_i\} \subseteq \{a_j\} \iff i = j \) and \( A_i^{\text{orderby}} \subseteq A_j^{\text{grid}} \iff \{a_i\} \subseteq \{a_j\} \iff i = j \).

In conclusion, using the presented transformation, a solver for the histogram selection problem is also able to solve the 0-1 knapsack problem. The chosen set of indexes of histograms of the histogram selection problem corresponds directly to the chosen set of indexes of the 0-1 knapsack problem. Because the presented construction can be done in linear-time, i.e., polynomial-time, it follows that 0-1 knapsack \( \leq^P \) HS.

The following lemma is needed in the proof of proposition 19.

**Lemma 18** Given variables \( x \in \text{var}(\mathbb{B}) \), \( y_j \in \text{var}(\mathbb{B}) \) and \( z_j \in \text{var}(\mathbb{B}) \) for all \( j \in \{1, \ldots, n\} \), the following statements are equivalent:

\[
(1) \quad x \iff \bigvee_{j=1}^n (y_j \land z_j)
\]
5. Pipelining

(2) \[ (1 - x + \sum_{j \in \{1, \ldots, n\} \wedge z_j} y_j \geq 1) \wedge \forall j \in \{1, \ldots, n\} : (z_j \Rightarrow (x + 1 - y_j \geq 1)) \]

Proof.

\[ x \Leftrightarrow \bigvee_{j=1}^{n} (y_j \wedge z_j) \]

(1) \[ x \Leftrightarrow \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} (y_j) \]

(2) \[ x \Rightarrow \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \wedge \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \]

(3) \[ \neg x \vee \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \wedge \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \]

(4) \[ \neg x \vee \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \wedge \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \]

(5) \[ \neg x \vee \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \wedge \bigvee_{j \in \{1, \ldots, n\} \wedge z_j} y_j \]

(6) \[ (1 - x) + \sum_{j \in \{1, \ldots, n\} \wedge z_j} y_j \geq 1 \]

\[ \wedge \forall j : ((j \in \{1, \ldots, n\} \wedge z_j) \Rightarrow (x + 1 - y_j \geq 1)) \]

(7) \[ (1 - x) + \sum_{j \in \{1, \ldots, n\} \wedge z_j} y_j \geq 1 \]

\[ \wedge \forall j \in \{1, \ldots, n\} : (z_j \Rightarrow (x + 1 - y_j \geq 1)) \]

The 0-1 ILP, a specialized case of the more general integer programming problem, is a mathematical optimization problem in which all of the variables are restricted to be 0 or 1. It is one of the famous Karp’s 21 NP-complete problems [Kar72].

Proposition 19 (HS solvable by ILP) The histogram selection problem (see definition 24) is solvable by the following ILP:

\[
\max \sum_{i=1}^{m} x_i \cdot b_i \\
\text{s.t. } \sum_{j=1}^{n} y_j \cdot s_j \leq S \\
\wedge \forall i \in \{1, \ldots, m\} : \left( 1 - x_i + \sum_{j \in \{1, \ldots, n\} \wedge A_i^{order} \subseteq \widehat{A} \text{ grid}, \wedge A_i \subseteq \widehat{A} \text{ alt} y_j \geq 1 \right) \\
\wedge \forall i \in \{1, \ldots, m\} : \forall j \in \{1, \ldots, n\} : 
\]
5.2. Histograms for the Estimation of Query Bounds

\[
\left( \left( A_i^{\text{orderby}} \subseteq A_j^{\text{grid}} \land A_i^{\text{where}} \subseteq A_j^{\text{all}} \right) \Rightarrow (x_i + 1 - y_j \geq 1) \right)
\]

\( \land \ 0 \leq x_i \leq 1 \ \text{for all} \ i \in \{1, \ldots, m\} \)

\( \land \ 0 \leq y_j \leq 1 \ \text{for all} \ j \in \{1, \ldots, n\} \)

Proof.

\[
\max_{J \subseteq \{1, \ldots, n\}} \max_{i \in \{1, \ldots, m\} \setminus J} \left( \sum_{j \in J} b_i \right) \\
\text{s.t.} \ \sum_{j \in J} s_j \leq S \\
\text{(1)} \ \Leftrightarrow \ \max_{i \in \{1, \ldots, m\}} \sum_{j=1}^n x_i \cdot b_i \\
\text{s.t.} \ \sum_{j=1}^n y_j \cdot s_j \leq S \\
\land \ \forall i \in \{1, \ldots, m\} : x_i \Leftrightarrow \bigvee_{j=1}^n (y_j \land za_{ij} \land zb_{ij}) \\
\land \ 0 \leq x_i \leq 1 \ \text{for all} \ i \in \{1, \ldots, m\} \\
\land \ 0 \leq y_j \leq 1 \ \text{for all} \ j \in \{1, \ldots, n\} \\
\text{with} \ za_{ij} := A_i^{\text{orderby}} \subseteq A_j^{\text{grid}} \ \text{and} \ zb_{ij} := A_i^{\text{where}} \subseteq A_j^{\text{all}} \\
\text{for all} \ i \in \{1, \ldots, m\} \ \text{and} \ j \in \{1, \ldots, n\} \\
\text{(2)} \ \Leftrightarrow \ \text{with lemma 18:} \\
\max_{i \in \{1, \ldots, m\}} \sum_{j=1}^n x_i \cdot b_i \\
\text{s.t.} \ \sum_{j=1}^n y_j \cdot s_j \leq S \\
\land \ \forall i \in \{1, \ldots, m\} : 1 - x_i + \sum_{j \in \{1, \ldots, n\} \setminus za_{ij} \land zb_{ij}} y_j \geq 1 \\
\land \ \forall i \in \{1, \ldots, m\} : \forall j \in \{1, \ldots, n\} : \\
\quad \left( (za_{ij} \land zb_{ij}) \Rightarrow (x_i + 1 - y_j \geq 1) \right) \\
\land \ 0 \leq x_i \leq 1 \ \text{for all} \ i \in \{1, \ldots, m\} \\
\land \ 0 \leq y_j \leq 1 \ \text{for all} \ j \in \{1, \ldots, n\} \\
\text{with} \ za_{ij} := A_i^{\text{orderby}} \subseteq A_j^{\text{grid}} \ \text{and} \ zb_{ij} := A_i^{\text{where}} \subseteq A_j^{\text{all}} \\
\text{for all} \ i \in \{1, \ldots, m\} \ \text{and} \ j \in \{1, \ldots, n\}
5. Pipelining

\begin{align*}
\text{(3) } & \max \sum_{i=1}^{m} x_i \cdot b_i \\
\text{s.t. } & \sum_{j=1}^{n} y_j \cdot s_j \leq S \\
\wedge \forall i \in \{1, \ldots, m\} & : \left(1 - x_i + \sum_{j \in \{1, \ldots, n\} \wedge A_{i}^{\text{orderby}} \subseteq A_{j}^{\text{grid}} \wedge A_{i}^{\text{where}} \subseteq A_{j}^{\text{all}}} y_j \geq 1\right) \\
\wedge \forall i \in \{1, \ldots, m\} : \forall j \in \{1, \ldots, n\} & : \left(A_{i}^{\text{orderby}} \subseteq A_{j}^{\text{grid}} \wedge A_{i}^{\text{where}} \subseteq A_{j}^{\text{all}}\right) \Rightarrow (x_i + 1 - y_j \geq 1) \\
\wedge 0 \leq x_i \leq 1 & \text{ for all } i \in \{1, \ldots, m\} \\
\wedge 0 \leq y_j \leq 1 & \text{ for all } j \in \{1, \ldots, n\}
\end{align*}

Therefore, we can conclude that the histogram selection problem is indeed NP-complete.

**Corollary 20 (HS is NP-complete)** The histogram selection problem (HS, see definition 24) is NP-complete, because 0-1 knapsack $\leq^P_m$ HS (see proposition 17) and HS $\leq^P_m$ ILP (see proposition 19 and its proof), and 0-1 knapsack and ILP are well-known NP-complete problems. [Kar72, Sah75]

As a direct consequence of this fact, the top-k semantic cache will have to heuristically decide which histograms to create and to maintain.

We can utilize a polynomial-time approximation scheme on the linear program of the histogram selection problem (see proposition 19) to find an approximate solution [DLHKW08].

Alternatively, we can apply a solver for mixed integer linear programming to it. The best implementations are the commercial solvers CPLEX [IBMa] and Gurobi [Gur]. But there exist open source alternatives. Examples are SCIP [Ach09], GLPK [Sch], and lp_solve [Thee]. In addition, the Computational Infrastructure for Operations Research (COIN-OR) [Thec] is an open source community that provides, for example, the solvers ABACUS [EGJR01], BCP [Theb], CBC [FHH+], and SYMPHONY [LRGM].
5.3. Estimating Query Bounds with Histograms

We want to estimate a lower bound \( \beta \) of the result of a given query \( q = (\ldots, \hat{q}, O, \ldots) \) (see definition 3). We assume that we have a maintained multidimensional histogram \( H^e = (A^{grid}, A^{all}, R, B, I) \) available, which is a histogram for the estimation of query bounds (see definition 21).

The tuple count \( \tau \) (see definition 20) describes the tuples in the bucket that are relevant for the query \( q \). In case of a remainder query (see section 6.2.4), the relevant tuple count is the number of tuples that are covered by the specific bucket, but have not yet been loaded by the top-k semantic cache.

5.3.1. The Idea

Algorithm 10 presents the idea of the algorithm getLowerBound(\( H^e, q, \tau \)). Naturally, it is very inefficient, because it does not use the index structure of the multidimensional histogram.

**Algorithm 10** getLowerBound(\( H^e, q, \tau \)) — idea (without index usage)

**Input:** histogram for the estimation of query bounds \( H^e = (A^{grid}, A^{all}, R, B, I) \), query \( q = (\ldots, \hat{q}, O, \ldots) \) and tuple count \( \tau \)

**Output:** lower bound \( \beta \)

1: \( \beta := \min_{\mu} \{ \mu \mid \exists b : (b = (l, u, t) \land b \in B \land \mu = \text{getLowerBound}(b, O) \land t(\tau) > 0 \land \text{overlapsBucket}(\hat{q}, b)) \} \) // see algorithm 11

2: return \( \beta \)

The algorithm regards all buckets of the histogram \( H^e \). Only buckets that are not empty are considered, i.e., \( t(\tau) > 0 \) must hold for the utilized tuple count \( \tau \). The algorithm checks if the bucket and the query overlap. All buckets that do not overlap with the query are dismissed. For each of the remaining buckets, a lower bound with respect to the order by \( O \) of query \( q \) is created. The lower bound \( \beta \) for the query \( q \) is the smallest created lower bound, again, with respect to the order by \( O \) of query \( q \).

In this version of the algorithm getLowerBound(\( H^e, q, \tau \)), which only presents the concept, two other methods are called, getLowerBound(\( b, O \)) and overlapsBucket(\( \hat{q}, b \)). But since both of them are also invoked in the actual realization of the algorithm getLowerBound(\( H^e, q, \tau \)), we will look at them reasonably closely.

The algorithm getLowerBound(\( b, O \)) (see algorithm 11) calculates the lower bound of a bucket \( b \) with respect to an order by \( O \).
Algorithm 11 getLowerBound(b, O)

Input: bucket \( b = (l, u, t) \) and ORDER BY clause \( O = ((a_1, d_1), \ldots, (a_{|O|}, d_{|O|})) \)

Output: lower bound \( \beta \)

1: \( \beta \in \times_{a \in \{a_1, \ldots, a_{|O|}\}} \text{dom}(a) \)
2: for each \( (a, d) \in O \) do
3: if \( a = \uparrow \) then
4: \( \beta[a] := l[a] \)
5: else
6: \( \beta[a] := u[a] \)
7: end if
8: end for
9: return \( \beta \)

It considers every component of the order by (see definition 11). If the component defines an ascending order, it will choose the value of the lower bound of the corresponding dimension of the bucket. On the other hand, if the component defines a descending order, it will take the value of the upper bound of the defined area of the bucket.

Example 25 (Lower bound of bucket \( b_5 \) for order by \( O_1 \)) Let \( b_5 \) be a bucket of a multidimensional histogram with

\[
   b_5 := \{(\text{id}, \text{likes}) = (16, 90), (\text{id}, \text{likes}) = (22, 160), t(\tau) = 4\}.
\]

Let \( O_1 = ((\text{likes}, \downarrow), (\text{id}, \uparrow)) \) be the order by of query \( q_1 \) (see table 4.4). Then algorithm getLowerBound(b_5, O_1) (see algorithm 11) determines the following lower bound \( \beta \) for bucket \( b_5 \) with respect to order by \( O_1 \):

\[
   \beta[\text{id}, \text{likes}] = (16, 160)
\]

The algorithm overlapsBucket(\( \hat{q}, b \)) (see algorithm 12) checks whether an expression \( \hat{q} \) of a query \( q \) overlaps with a bucket \( b \). The algorithm could just use an SMT solver to determine whether there was an overlap. And it will do so, if necessary, by calling Solver.overlaps(\( \hat{q}, b \)). The condition \( \hat{b} \) is the bucket condition. A condition that describes the hyperrectangle that is spanned by the bucket can easily be calculated using the lower and upper bound of the bucket. But before calling the SMT solver, the algorithm will try a shortcut. It will choose a number of representative tuples \( C \) that are contained within the bucket \( b \). If the condition \( \hat{q} \) evaluates to true for any of these tuples when their values are inserted into it, the algorithm can immediately conclude that the expression \( \hat{q} \) overlaps with the bucket \( b \). Of course, we cannot reasonably check all corners of the hyperrectangle of the bucket, because an \( n \)-dimensional bucket has \( 2^n \) corners. In addition, checking all corners would
5.3. Estimating Query Bounds with Histograms

Algorithm 12 overlapsBucket($\hat{q}$, $b$)

**Input:** expression $\hat{q}$ and bucket $b$  
**Output:** true iff. expression $\hat{q}$ and bucket $b$ overlap

1: $C := \text{chooseTuples}(b)$ // see algorithm 13
2: if $\exists \mu \in C : \hat{q}[^{\mu}] \equiv \text{true}$ then
3: return true
4: else
5: return Solver.overlaps($\hat{q}, \hat{b}$)
6: end if

not even be enough. It is possible that an expression $\hat{q}$ overlaps with a bucket $b$, but it does not overlap with any of the corners of the bucket. Therefore, this approach is just a heuristic. But it turns out to be an extremely effective one in practice (see example 26). One possible implementation of the algorithm chooseTuples($b$) is choosing the lower and upper bound of the bucket and calculating the mean tuple of the bucket that sits at the center of the spanned area of the bucket (see algorithm 13). This is the implementation used in IQCache, our prototype of the top-k semantic cache.

Algorithm 13 chooseTuples($b$)

**Input:** bucket $b$  
**Output:** set of interesting tuples of the bucket

1: return \{l, u, mean(l, u)\}

Example 26 (Overlap of bucket $b_T^5$ and expression $\hat{q}_1$) Let $b_T^5$ be a bucket of a multi-dimensional histogram. It is defined as follows:

\[
b_T^5 := (l, u, t) \quad \text{with}
\]

\[
l[^{id, likes, created}] = (16, 90, \text{today}),
\]

\[
u[^{id, likes, created}] = (22, 160, \text{before_last_hour}) \quad \text{and}
\]

\[
t(\tau) = 1.
\]

Let the expression $\hat{q}_1$ of query $q_1$ be defined as follows (see table 4.4):

\[
\hat{q}_1 = (\text{created} \geq \text{last_week}) \land (\text{likes} \geq 100)
\]

We can state the bucket condition $\hat{b}_T^5$ as follows:

\[
\hat{b}_T^5 := ((16 \leq \text{id}) \land (\text{id} \leq 22))
\]

\[
\land ((90 \leq \text{likes}) \land (\text{likes} \leq 160))
\]

\[
\land ((\text{today} \leq \text{created}) \land (\text{created} \leq \text{before_last_hour}))
\]
5. Pipelining

Bucket $b^T_5$ and expression $\hat{q}_1$ overlap, because $\hat{b}^T_5 \land \hat{q}_1$ is satisfiable, which can be determined by an SMT solver (see proposition 7).

But we can also just insert the lower bound $l$ and the upper bound $u$ of bucket $b^T_5$ into expression $\hat{q}_1$. We start by inserting the lower bound $l$:

$$\hat{q}_1[l] = ((\text{created} \geq \text{last week}) \land (\text{likes} \geq 100))[l]$$

$$= (\text{today} \geq \text{last week}) \land (90 \geq 100)$$

$$\equiv \text{true} \land \text{false}$$

$$\equiv \text{false}$$

An evaluation to false does not help us at all, because it means nothing. Hence, we insert the upper bound $u$ into expression $\hat{q}_1$:

$$\hat{q}_1[u] = ((\text{created} \geq \text{last week}) \land (\text{likes} \geq 100))[u]$$

$$= (\text{before last hour} \geq \text{last week}) \land (160 \geq 100)$$

$$\equiv \text{true} \land \text{true}$$

$$\equiv \text{true}$$

Because the expression is evaluated to true, we can conclude that the bucket $b^T_5$ and expression $\hat{q}_1$ do overlap. A call to an SMT solver is not necessary.

5.3.2. A Sample Multidimensional Histogram

We exemplify the algorithm for the estimation of a lower bound for the result of a query using the three queries $q_1$, $q_2$ and $q_3$ as well as the sample instance from our running example (see section 1.2.1 and table 1.1).

In addition, we need a sample multidimensional histogram of relation $post$ based on its sample instance.

**Example 27 (Sample multidimensional histogram)** Figure 5.1 shows the sample multidimensional histogram $H^c$ that we will use in this chapter. It is a histogram for the estimation of query bounds (see definition 21). The buckets for the attributes $id$ and $likes$ are arranged in a grid structure. Therefore, as we will understand later in this chapter, the histogram supports all possible attribute-based lexicographical orders based on these two attributes. And the attribute set $A^{grid}$ contains the following attributes:

$$A^{grid} = \{id, likes\}$$

The dimension created does not need to follow the grid structure, but it is part of the multidimensional histogram. Hence, the attribute set $A^{all}$ consists of the following attributes:

$$A^{all} = \{id, likes, created\}$$
The underlying relation of the histogram certainly is the relation \( \text{post} \), i.e., \( R = \{\text{post}\} \). The buckets \( B \) of the histogram are shown in figure 5.1. And we assume that an index structure \( I \) exists.

![Figure 5.1. A sample histogram for the tuples from the running example](image)

After the execution of a query, if the query could not be completely answered by the cache and tuples needed to be transferred from the server, the tuple counts of the histogram for the estimation of query bounds have to be updated.

**Example 28 (Updated histogram after execution of query \( q_1 \))** At the start, the top-k semantic cache is empty. As a consequence of this, when query \( q_1 \) is processed, there
are no segments in the top-k semantic cache that overlap with query $q_1$. Hence, the complete result of query $q_1$ must be transferred from the server. (The remainder query $r_1$ is exactly the same as query $q_1$.)

The histogram for the estimation of query bounds must be updated. The tuple count that describes the number of tuples that have not yet been loaded in the affected buckets must be reduced accordingly. Figure 5.2 shows the updated histogram.

Please note that after the update, the buckets $b_1$, $b_2$ and $b_3$ are empty, i.e., have a tuple count of 0. We could delete these buckets for now. But we will allow buckets with a tuple count of 0 to exist. Because cache replacement would result in the need to recreate these or similar buckets, anyway. These constant updates to the structure of
the histogram would produce a huge processing overhead for the top-k semantic cache. In addition, there may be another relevant non-zero tuple count present in the bucket that prevents deletion of the bucket.

5.3.3. The Algorithm

We have an idea what the algorithm for the estimation of query bounds strives to do (section 5.3.1). And we expanded our running example by a sample multidimensional histogram (see section 5.3.2). Henceforth, we will discuss the actual realization of the algorithm getLowerBound($H^e, q, \tau$) (see algorithm 14), which uses the index structure of the multidimensional histogram.

Algorithm 14 getLowerBound($H^e, q, \tau$)

| Input: histogram for estimation of query bound $H^e = (A^{grid}, A^{all}, R, B, I)$, query $q = (\ldots, \hat{q}, O, \ldots)$ and tuple count $\tau$ |
| Output: lower bound $\beta$ |
| 1: $r := (l_H, u_H)$ |
| 2: $b := \text{getSmallestBucket($H^e, q, \tau, 1, r$)}$ // see algorithm 15 |
| 3: return getLowerBound($b, O$) // see algorithm 11 |

The algorithm creates a region $r$ that describes the relevant area of the histogram, which is a hyperrectangle. It is set to $(l_H, u_H)$. $l_H$ is the lower bound and $u_H$ is the upper bound of the histogram itself. Hence, initially, the region $r$ spans the whole histogram. Then, the smallest relevant bucket, i.e., the smallest bucket with non-zero tuple count that overlaps with the query condition, is determined by algorithm 15. The lower bound $\beta$ of the smallest bucket is calculated using algorithm 11 which we have already discussed (see section 5.3.1) and, finally, returned.

The algorithm getSmallestBucket($H^e, q, \tau, i, r$) (see algorithm 15) is a recursive algorithm that finds the smallest bucket of the histogram $H^e$ that overlaps with query $q$ and has a positive tuple count $\tau$. The parameter $i$ denotes the current order by component of the order by $O$. In the call of getSmallestBucket($H^e, q, \tau, i, r$) by algorithm getLowerBound($H^e, q, \tau$), the parameter $i$ has thus been set to the first component of the order by. The final parameter $r$ is the considered region within the histogram. First, the algorithm checks if the exit condition $e$ of the recursion is met (see algorithm 16). This method might find a single bucket $b$ in the region $r$. In that case, the bucket $b$ is returned. Secondly, the region $r$ is split into a sequence $S$ of sub-regions that overlap with the query condition $\hat{q}$ based on the ordering of the order by component at position $i$ of the order by $O$ (see algorithm 18). The current order by component might change to position $j$ as a result of this call.
5. Pipelining

Algorithm 15 getSmallestBucket($H^e$, $q$, $\tau$, $i$, $r$)

**Input:** histogram for estimation of query bound $H^e = (A^\text{ord}, A^\text{all}, R, B, I)$, query $q = (...)\hat{q}, O, ...$, tuple count $\tau$, index $i$ of current component of the order by and rectangle $r$

**Output:** smallest bucket $b$

1. $(e, b) := \text{exitCondition}(H^e, q, \tau, r)$ \hspace{1cm} // see algorithm 16
2. if $e \equiv \text{true}$ then
3. return $b$
4. end if
5. $(j, S) := \text{chooseRegions}(H^e, q, i, r)$ \hspace{1cm} // see algorithm 18
6. if $S \neq []$ then
7. return $\text{processRegions}(H^e, q, \tau, j, S)$ \hspace{1cm} // see algorithm 20
8. else
9. return $\text{selectBucket}(H^e, q, \tau, r)$ \hspace{1cm} // see algorithm 22
10. end if

Thirdly, if the sequence $S$ of sub-regions is not empty, algorithm $\text{processRegions}(H^e, q, \tau, j, S)$ (see algorithm 20) will then process the sequence $S$ of sub-regions. This algorithm contains the recursive call to getSmallestBucket($H, q, \tau, j, s$).

On the other hand, if the sequence $S$ of sub-regions is an empty sequence, algorithm $\text{selectBucket}(H^e, q, \tau, r)$ (see algorithm 22) will be used to inspect all buckets of the current region $r$ to find the smallest one.

The algorithm $\text{exitCondition}(H^e, q, \tau, r)$ (see algorithm 16), which implements the exit condition of the recursive algorithm getSmallestBucket($H^e, q, \tau, i, r$), starts off by scanning the region $r$ with algorithm $\text{scanRegion}(H^e, \tau, r)$ (see algorithm 17). That algorithm returns the pair $(n, b)$, which represents the result of the scan.

The value of $n$ can be either 0, 1 or 2. It will be 0 if no bucket that has a positive tuple count $\tau$ has been found in the region $r$. It will be 1 if exactly one bucket that has a positive tuple count $\tau$ has been found in region $r$. And it will be 2 otherwise. If exactly one bucket that has a positive tuple count $\tau$ has been found in the region $r$, $b$ will be that bucket. Otherwise, $b$ will be $\perp$.

If $n$ is 0 or 1, the exit condition will be fulfilled. There is no need to split the region into sub-regions. If $n$ is 0, no relevant bucket has been found in the region $r$. If $n$ is 1, we have to find out if the one bucket $b$ that has been found in the region $r$ is relevant for the query $q$, i.e., we have to check if the bucket $b$ overlaps with the query condition $\hat{q}$.

The algorithm $\text{scanRegion}(H^e, \tau, r)$ (see algorithm 17) checks whether or not a region contains zero, one or more than one bucket that has a positive tuple count $\tau$. The algorithm will return the result of this scan. In addition, if the region $r$ contains exactly one such bucket, it will return it as well.
5.3. Estimating Query Bounds with Histograms

Algorithm 16 \texttt{exitCondition}(H^e, q, \tau, r)

\textbf{Input:} histogram for estimation of query bound $H^e = (A^{grid}, A^{all}, R, B, I)$, query $q = (\ldots, \hat{q}, O, \ldots)$, tuple count $\tau$ and rectangle $r$

\textbf{Output:} pair $(e, b)$ with $e$ is true iff. the region $r$ contains exactly one or no buckets, and $b$ will contain the only bucket of the region $r$ if it exists

\begin{algorithmic}
\State $(n, b) := \texttt{scanRegion}(H^e, \tau, r)$ \Comment{see algorithm 17}
\If{$n = 0$}
\State \textbf{return} $(\text{true}, \bot)$
\ElsIf{$n = 1$}
\If{$\texttt{overlapsBucket}(\hat{q}, b)$} \Comment{see algorithm 12}
\State \textbf{return} $(\text{true}, b)$
\Else
\State \textbf{return} $(\text{true}, \bot)$
\EndIf
\Else
\State \textbf{return} $(\text{false}, \bot)$
\EndIf
\end{algorithmic}

Algorithm 17 \texttt{scanRegion}(H^e, \tau, r)

\textbf{Input:} histogram for estimation of query bound $H^e = (A^{grid}, A^{all}, R, B, I)$, tuple count $\tau$ and region $r = (l, u)$

\textbf{Output:} pair $(n, b)$ with $n := 0$ and $b := \bot$ if no bucket that has a positive tuple count has been found in the region $r$, with $n := 1$ and $b$ the bucket if exactly one bucket that has a positive tuple count has been found in the region $r$, and with $n := 2$ and $b := \bot$ if more than one bucket that has a positive tuple count has been found in the region $r$

\begin{algorithmic}
\If{$\exists b_1 \exists b_2 : (b_1 = (l_1, u_1, t_1) \land b_2 = (l_2, u_2, t_2) \land b_1 \in B[r] \land b_2 \in B[r] \land b_1 \neq b_2 \land t_1(\tau) > 0 \land t_2(\tau) > 0)$}
\State \textbf{return} $(2, \bot)$
\ElsIf{$\exists b : (b = (l, u, t) \land b \in B[r] \land t(b) > 0)$}
\State \textbf{return} $(1, \varepsilon b : (b = (l, u, t) \in B[r] \land t(\tau) > 0))$
\Else
\State \textbf{return} $(0, \bot)$
\EndIf
\end{algorithmic}
5. Pipelining

The runtime of algorithm scanRegion\( (H^e, \tau, r) \) is negligibly small, because it does not need to invoke an SMT solver. In fact, the used index structure \( I \) of histogram \( H^e \) might even contain aggregate tuple counts. In that case, the execution of the algorithm only requires constant time.

The task of algorithm chooseRegions\( (H^e, q, i, r) \) (see algorithm [18]) is to split the current region \( r \) into a sequence \( S \) of sub-regions with respect to the order by \( O \). The sub-regions in the returned sequence are sorted with respect to the order by \( O \). This sorting is needed to ensure that the algorithm getSmallestBucket\( (H^e, q, \tau, i, r) \) (see algorithm [15]) actually returns the smallest bucket of the histogram.

**Algorithm 18** chooseRegions\( (H^e, q, i, r) \)

**Input:** histogram for estimation of query bound \( H^e = (A^{grid}, A^{all}, R, B, I) \), query \( q = (\ldots, \tilde{q}, O, \ldots) \) with order by clause \( O = ((a_1, d_1), \ldots, (a_{|O|}, d_{|O|})) \), index \( i \) of current component of the order by and rectangle \( r \)

**Output:** pair \( (j, S) \) of index \( j \) of new current component of the order by and sequence \( S = [s_1, \ldots, s_{|S|}] \) of regions

1: \( j := i \)
2: while \( j \leq |O| \wedge \neg \text{splitable}(r, a_j) \) do
3: \( j := j + 1 \)
4: end while
5: \( c := \perp \)
6: if \( j \leq |O| \) then
7: \( c := a_j \)
8: else
9: \( A^{orderby} := \{a_1, \ldots, a_{|O|}\} \)
10: \( c := \varepsilon a : ((a \in A^{all} \setminus A^{orderby}) \wedge \text{splitable}(r, a)) \)
11: end if
12: if \( c \) is \( \perp \) then
13: return \( (\perp, []) \)
14: end if
15: \( (r_l, r_u) := \text{splitRegion}(r, c) \) // see algorithm [19]
16: if \( j > |O| \vee d_j = \uparrow \) then
17: return \( (j, [r_l, r_u]) \)
18: else
19: \( (j, [r_u, r_l]) \)
20: end if

The order by \( O \) must thereby be an attribute-based lexicographical ordering (see definitions [11] and [12]).

At first, the region is split – with every call of the algorithm chooseRegions\( (H^e, q, i, r) \)
5.3. Estimating Query Bounds with Histograms

– using the dimension, i.e., attribute, of the first order by component until it cannot be split further along this dimension. Thereafter, the algorithm moves on to the attribute of the second order by component and only utilizes this dimension until the region cannot be split further. Then, the algorithm selects the next one and so on and so forth. It works this way to actually find the smallest bucket with respect to the order by $O$ and not just any bucket that happens to have a positive tuple count and overlaps with the query condition.

If all attributes of $A_{\text{orderby}}$ have been used, the algorithm chooses a random attribute of the set $A_{\text{all}} \setminus A_{\text{orderby}}$.

To work, the approach of splitting according to the order by has demands on the index structure of the histogram $H^e$. The index structure must support splitting regions into sub-regions using the dimensions in the same order as they are present in components of the order by.

Generally, the algorithm requires that the histogram has a phased [MD88] structure for the dimensions utilized in the order by that is compatible with the definition of that order by. For the rest of the attributes, the histogram may have any structure, mhist-p [PI97, II11] or even STHoles [BCG01] are perfectly acceptable.

As a consequence of this, if we want to support arbitrary attribute-based lexicographical orderings in the order bys, the buckets must be arranged in a grid structure along the dimensions that we want to allow to be used in the order by.

The algorithm splitRegion($r, a$) splits the region $r$ into two sub-regions $r_l$ and $r_u$ using the dimension $a$.

<table>
<thead>
<tr>
<th>Algorithm 19 splitRegion($r, a$)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> region $r = (l, u)$ that is splittable using dimension $a$, dimension / attribute $a$</td>
</tr>
<tr>
<td><strong>Output:</strong> a pair $(r_l, r_u)$ of two sub-regions of the region $r$ that has been split by dimension $a$</td>
</tr>
</tbody>
</table>

1: $l_l := l$
2: $u_l := u$
3: $l_u := l$
4: $u_u := u$
5: $v_{\text{min}} := l[a]$
6: $v_{\text{max}} := u[a]$
7: $v_l := \text{between}(v_{\text{min}}, v_{\text{max}})$
8: $v_u := \text{next}(v_u)$
9: $u_l[a] := v_l$
10: $u_u[a] := v_u$
11: \textbf{return} $(r_l := (l_l, u_l), r_u := (l_u, u_u))$

The minimum value $v_{\text{min}}$ of the dimension $a$ that will be split can be found in the lower
5. Pipelining

bound \( l \) of region \( r = (l, u) \), i.e., \( v_{\min} := l[a] \). On the other hand, the maximum value \( v_{\max} \) is contained in the upper bound \( u \), i.e., \( v_{\max} := u[a] \). Because the region \( r \) is splittable using dimension \( a \), we can find values \( v_l \) and \( v_u \) that are compatible with the index structure \( I \) such that \( v_{\min} \leq v_l < v_u \leq v_{\max} \) holds. They split the region \( r \) into the following sub-regions \( r_l = (l_l, u_l) \) and \( r_u = (l_u, u_u) \):

\[
\begin{align*}
  l_l[c] &:= l[c] \\
  u_l[c] &:= \begin{cases} 
    u[c] & \text{if } c \neq a \\
    v_l & \text{if } c = a 
  \end{cases} \\
  l_u[c] &:= \begin{cases} 
    l[c] & \text{if } c \neq a \\
    v_u & \text{if } c = a 
  \end{cases} \\
  u_u[c] &:= u[c]
\end{align*}
\]

Therefore, the regions \( r_l \) and \( r_u \) only differ in dimension \( a \). Using an ascending order, the interval \( [l_l[a], u_l[a]] \) is strictly before the interval \( [l_u[a], u_u[a]] \). The algorithm always returns \( r_l \) as the first and \( r_u \) as the second component of the resulting pair.

The algorithm processRegions(\( H^e, q, \tau, j, S \)) (see algorithm 20) processes all sub-regions \( s \) in the sequence \( S \).

**Algorithm 20** processRegions(\( H^e, q, \tau, j, S \))

**Input:** histogram for estimation of query bound \( H^e = (A^{grid}, A^{all}, R, B, I) \), query \( q = (\ldots, \hat{q}, O, \ldots) \), tuple count \( \tau \), index \( j \) of current component of the order by and sequence \( S = [s_1, \ldots, s_{|S|}] \) of regions

**Output:** smallest bucket \( b \)

1: for each \( s \in S \) do
2: if overlapsRegion(\( \hat{q}, s \)) then \hspace{1cm} // see algorithm 21
3: \( b := \) getSmallestBucket\( (H, q, \tau, j, s) \) \hspace{1cm} \hspace{1cm} // recursive call, see algorithm 15
4: if \( b \) is not \( \bot \) then
5: return \( b \)
6: end if
7: end if
8: end for
9: return \( \bot \)

If a current sub-region \( s \) overlaps with the query condition \( \hat{q} \), which is determined by algorithm overlapsRegion(\( \hat{q}, s \)) (see algorithm 21), the algorithm getSmallestBucket\( (H, q, \tau, j, s) \) (see algorithm 15) will be recursively called. Otherwise, the current sub-region \( s \) is discarded.

If the recursive call to getSmallestBucket\( (H, q, \tau, j, s) \) returns a bucket \( b \) for a sub-region \( s \), no other sub-regions will be considered. The bucket \( b \) will be returned and the algorithm
will have finished. On the other hand, if the recursive call does not yield a bucket \( b \), but returns \( \bot \) instead, the next sub-region \( s \) will be processed. It is possible that no sub-region finds a relevant bucket \( b \). If that is the case, the algorithm will return \( \bot \) as well. This means that there are no tuples at the server that are relevant to the query. Consequently, all cached tuples can be pipelined.

The algorithm \texttt{overlapsRegion}(\( \tilde{q}, r \)) (see algorithm 21) works exactly like the algorithm \texttt{overlapsBucket}(\( \tilde{q}, b \)) (see algorithm 12).

\begin{algorithm}
\caption{overlapsRegion(\( \tilde{q}, r \))}
\begin{algorithmic}[1]
  \State \textbf{Input:} expression \( \tilde{q} \) and region \( r \)
  \State \textbf{Output:} true iff. expression \( \tilde{q} \) and region \( r \) overlap
  \State 1: \( C := \text{chooseTuples}(r) \) \hfill // see algorithm 13
  \State 2: \textbf{if} \( \exists \mu \in C : \tilde{q}[\mu] \equiv \text{true} \) \textbf{then}
  \State 3: \textbf{return} \text{true}
  \State 4: \textbf{else}
  \State 5: \textbf{return} \text{Solver.overlaps}(\( \tilde{q}, \tilde{r} \))
  \State 6: \textbf{end if}
\end{algorithmic}
\end{algorithm}

Similarly, it will select a set of representative tuples \( C \) that are contained within the region \( r \). If the query condition \( \tilde{q} \) evaluates to true for any of these tuples, the algorithm can skip the SMT solver and can instantly return true. But should the query condition \( \tilde{q} \) evaluate to false for all tuples in \( C \), the algorithm will use an SMT solver to check whether or not the query \( q \) and the region \( r \) overlap. To do this, a region condition \( \tilde{r} \) that describes the region \( r \) is created for the region.

Finally, the algorithm \texttt{selectBucket}(\( H^e, q, \tau, r \)) (see algorithm 22) finds the smallest bucket \( b \) with respect to the \textit{order by} \( O \) in a specified region \( r \).

The algorithm is basically the operational implementation of the bucket selection in algorithm \texttt{getLowerBound}(\( H^e, q, \tau \)) (see algorithm 10). It iterates all buckets \( b \) of the region \( r \). If the bucket \( b \) has a positive tuple count \( \tau \) and overlaps with the query condition \( \tilde{q} \), its lower bound will be considered. In the end, the algorithm returns the bucket with the smallest of all considered lower bounds with respect to the \textit{order by} \( O \).
5. Pipelining

**Algorithm 22** selectBucket($H^e$, $q$, $\tau$, $r$)

**Input:** histogram for estimation of query bound $H^e = (A^{\text{grid}}, A^{\text{all}}, R, B, I)$, query $q = (\ldots, \hat{q}, O, \ldots)$, tuple count $\tau$ and rectangle $r$

**Output:** smallest bucket $c$

1. $c := \bot$
2. for each $b = (l, u, t) \land b \in B[r]$ do
3.  
4.  if $t(\tau) > 0$ then
5.  
6.  if overlapsBucket($\hat{q}, b$) then // see algorithm 12
7.  
8.  if $c$ is $\bot \lor b < O$ c then
9.  
10.  // $b < O$ c iff. getLowerBound($b, O$) < $O$ getLowerBound($c, O$) // see algorithm 11
11.  
12.  $c := b$
13.  end if
14. end for
15. return $c$

5.3.4. Two Examples

**Example 29** (Lower bound for remainder query $r_2$ of query $q_2$)

We used the following remainder query $r_2$ for query $q_2$ (see example 17):

$$r_2 := q_2 \setminus (\pi_{A^e_{2 \text{select}}} (\sigma_{\hat{i}_2} (x_{r \in R_2 r})))$$

If we ignore the limit of query $q_2$ in the remainder query $r_2$, we can formulate the following remainder condition $\hat{r}_2$:

$$\hat{r}_2 := \hat{q}_2 \land (\neg \hat{i}_2 \lor u \hat{i}_2)$$

The sub-expressions $\hat{q}_2$ and $\hat{i}_2$ have already been computed (see table 4.8):

$$\hat{q}_2 = (\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last hour}) \lor (\text{likes} \geq 20))$$
$$\hat{i}_2 = (\text{created} \geq \text{last week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$$

Using these expressions, we can calculate the remainder condition $\hat{r}_2$ as follows:

$$\hat{r}_2 := \hat{q}_2 \land (\neg \hat{i}_2 \lor u \hat{i}_2)$$
$$= (\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last hour}) \lor (\text{likes} \geq 20)) \land (\neg ((\text{created} \geq \text{last week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))) \lor u((\text{created} \geq \text{last week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$
$$\equiv (\text{created} \geq \text{today}) \land ((\text{created} \geq \text{last hour}) \lor (\text{likes} \geq 20)) \land (\neg ((\text{created} \geq \text{last week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))) \lor u((\text{created} \geq \text{last week}) \land ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))))$$
5.3. Estimating Query Bounds with Histograms

\textbf{Figure 5.3} shows the histogram $H^e$ before processing of query $q_2$ and before execution of the remainder query $r_2$. The remainder condition $\hat{r}_2$ is visualized in the diagram.

We use algorithm \texttt{getLowerBound($H^e$, $r_2$, server)} (see algorithm 14) to calculate a lower bound for the remainder query $r_2$:

\begin{verbatim}
1   \rightarrow getLowerBound($H^e$, $r_2$, server) \// see algorithm 14
2   s := ((1, 10, last_week), (28, 250, now))
3   \rightarrow getSmallestBucket($H^e$, $r_2$, server, 1, s) \// see algorithm 15
4   \rightarrow exitCondition($H^e$, $r_2$, server, s) \// see algorithm 16
5   \leftarrow (false, \bot)
6   \rightarrow chooseRegions($H^e$, $r_2$, server, 1, s) \// see algorithm 18
\end{verbatim}

131
5. Pipelining

\[
(1, S := [(1, 10, \text{last week}), (15, 250, \text{now})], \\
s_2 := [(16, 10, \text{last week}), (28, 250, \text{now})])
\]

→ processRegions(\(H^e, r_2, \text{server}, 1, S\)) // see algorithm 20
→ overlapsRegion(\(\hat{r}_2, s_1\)) // see algorithm 21
← true
→ getSmallestBucket(\(H^e, r_2, \text{server}, s_1\))
→ exitCondition(\(H^e, r_2, \text{server}, s_1\))
← (false, ⊥)
→ chooseRegions(\(H^e, r_2, \text{server}, s_1\))
← (1, \(S_1 := [s_{11} := ((1, 10, \text{last week}), (9, 250, \text{now})], \\
s_{12} := ((10, 10, \text{last week}), (15, 250, \text{now})])\)
→ processRegions(\(H^e, r_2, \text{server}, 1, S_1\))
→ overlapsRegion(\(\hat{r}_2, s_{11}\))
← true
→ getSmallestBucket(\(H^e, r_2, \text{server}, s_{11}\))
→ exitCondition(\(H^e, r_2, \text{server}, s_{11}\))
← (true, ⊥)
→ overlapsRegion(\(\hat{r}_2, s_{12}\))
← true
→ getSmallestBucket(\(H^e, r_2, \text{server}, s_{12}\))
→ exitCondition(\(H^e, r_2, \text{server}, s_{12}\))
← (true, b_6)
← b_6
← b_6
← b_6
→ getLowerBound(b_6, \(O_2 = ((\text{id}, \uparrow))\)) // see algorithm 11
← (10, ⊥, \ldots, ⊥)
← (10, ⊥, \ldots, ⊥)

At the beginning, the algorithm getLowerBound(\(H^e, r_2, \text{server}\)) (see algorithm 14) initializes the considered region, which we call region \(s\) instead of region \(r\) to avoid confusion with remainder query \(r_2\). Initially, the considered region \(s\) covers the whole histogram, i.e., \(s := ((1, 10, \text{last week}), (28, 250, \text{now}))\). Therefore, all buckets, i.e., \(b_1, \ldots, b_6\), are contained within the considered region \(s\).

Then, it calls algorithm getSmallestBucket(\(H^e, r_2, \text{server}, 1, s\)) (see algorithm 15).
5.3. Estimating Query Bounds with Histograms

Figure 5.3.: Histogram before processing of query $q_2$
5. Pipelining

This algorithm checks the exit condition by calling exitCondition($H^e$, $r_2$, server, $s$) (see algorithm 16). Contained within the considered region $s$, there are more than two buckets with a positive tuple count, namely $b_4$, $b_5$, $b_6$, $b_7$ and $b_8$. Hence, the exit condition is not met. Consequently, the region $s$ is split into sub-regions $S := [s_1, s_2]$ using algorithm chooseRegions($H^e$, $r_2$, server, 1, $s$) (see algorithm 18). The remainder query $r_2$ uses the order by $O_2 = ((\text{id}, \uparrow))$. Therefore, region $s$ is split along dimension id. The sub-region $s_1 := ((1,10,\text{last week}), (15,250,\text{now}))$ contains the buckets $b_1$, $b_2$, $b_4$ and $b_6$. On the other hand, the sub-region $s_2 := ((16,10,\text{last week}), (28,250,\text{now}))$ covers the buckets $b_3$, $b_5$, $b_7$ and $b_8$.

Algorithm processRegions($H^e$, $r_2$, server, 1, $S$) (see algorithm 20) processes the sub-regions $s_1$ and $s_2$. It starts with the smaller region $s_1$. The remainder query $r_2$ and region $s_1$ overlap (see figure 5.3), which can be determined by algorithm overlapsRegion($r_2$, $s_1$) (see algorithm 21). Thus, algorithm getSmallestBucket($H^e$, $r_2$, server, 1, $s_1$) is called recursively.

Again, first, the algorithm checks the exit condition exitCondition($H^e$, $r_2$, server, $s_1$). But still two buckets with a positive tuple count, namely buckets $b_4$ and $b_5$, are contained within the considered region $s_1$. The exit condition is not met. And the region $s_1$ is split into sub-regions $S_1 := [s_{11}, s_{12}]$ using algorithm chooseRegions($H^e$, $r_2$, server, $s_1$). The split once again uses the dimension id, because the remainder query $r_2$ uses the order by $O_2 = ((\text{id}, \uparrow))$. The sub-region $s_{11} := ((1,10,\text{last week}), (9,250,\text{now}))$ contains the buckets $b_1$ and $b_4$. And the sub-region $s_{12} := ((10,10,\text{last week}), (15,250,\text{now}))$ includes the buckets $b_3$ and $b_6$.

Algorithm processRegions($H^e$, $r_2$, server, 1, $S_1$) processes the sub-regions $s_{11}$ and $s_{12}$. It begins with the smaller region $s_{11}$. The region $s_{11}$ overlaps with the remainder query $r_2$ (see figure 5.3). Therefore, algorithm getSmallestBucket($H^e$, $r_2$, server, 1, $s_{11}$) is called recursively.

Algorithm getSmallestBucket($H^e$, $r_2$, server, 1, $s_{11}$) checks the exit condition. The algorithm exitCondition($H^e$, $r_2$, server, $s_{11}$) discovers that the region $s_{11}$ only contains one bucket with positive tuple count: the bucket $b_4$. But bucket $b_4$ does not contain any tuple that were created today or within the last hour (see figure 5.3). Hence, bucket $b_4$ and the remainder query $r_2$ do not overlap. As a consequence, algorithm exitCondition($H^e$, $r_2$, server, $s_{11}$) returns that the exit condition has been met, but no relevant buckets have been found within region $s_{11}$. And algorithm getSmallestBucket($H^e$, $r_2$, server, 1, $s_{11}$) returns ⊥.

Algorithm processRegions($H^e$, $r_2$, server, 1, $S_1$) considers the other region $s_{12}$. The remainder query $r_2$ and region $s_{12}$ also overlap (see figure 5.3). Again, algorithm getSmallestBucket($H^e$, $r_2$, server, 1, $s_{12}$) is called recursively.

The algorithm checks its exit condition by calling exitCondition($H^e$, $r_2$, server, $s_{12}$). The region $s_{12}$ contains only one bucket with a positive tuple count, the bucket $b_6$. The remainder query $r_2$ and bucket $b_6$ do not overlap. Finally, the smallest bucket has been found. It is bucket $b_6$. 134
The last step of algorithm getLowerBound($H^e$, $r_2$, server) is the creation of a lower bound for bucket $b_6$ based on the bounds of the bucket and the order by $O_2 = ((id, \uparrow))$ of the remainder query $r_2$ with algorithm getLowerBound($b_6$, $O_2 = ((id, \uparrow))$ (see algorithm 11). That algorithm returns the tuple $\beta_2 = (10, \perp, \ldots, \perp)$.

The result of algorithm getLowerBound($H^e$, $r_2$, server) is that all tuples that are transferred from the server using the remainder query $r_2$ are greater than or equal to the tuple $\beta_2 = (10, \perp, \ldots, \perp)$ with respect to the order by $O_2 = ((id, \uparrow))$ of the remainder query $r_2$. Hence, the id of these tuples is equal to or greater than 10 ($\beta_2[id]$).

The overlap of query $q_2$ and segment $s_1$ contains the tuples $\mu_7$, $\mu_8$, $\mu_9$, $\mu_{10}$, $\mu_{14}$ and $\mu_{19}$ (see table 5.7).

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>...</th>
<th>$q_1$</th>
<th>$q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>$\mu_7$</td>
<td>7</td>
<td>220</td>
<td>today</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$\mu_8$</td>
<td>8</td>
<td>180</td>
<td>today</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>$\mu_9$</td>
<td>9</td>
<td>170</td>
<td>last hour</td>
<td>...</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>$\mu_{10}$</td>
<td>10</td>
<td>230</td>
<td>last hour</td>
<td>...</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>$\mu_{14}$</td>
<td>14</td>
<td>190</td>
<td>today</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>$\mu_{19}$</td>
<td>19</td>
<td>200</td>
<td>last hour</td>
<td>...</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.7.: Tuples of the overlap of query $q_2$ and segment $s_1$

The tuples $\mu_7$, $\mu_8$ and $\mu_9$ have an id less than 10. As a consequence of the result of algorithm getLowerBound($H^e$, $r_2$, server), they can be returned immediately. The tuple $\mu_{10}$ has an id of exactly 10. Since the order by defines a total ordering and all tuples that are transferred from the server are greater than or equal to id 10, the tuple $\mu_{10}$ can be pipelined as well. In conclusion, the top-k semantic cache does not have to wait for the first tuple of the remainder query to return the tuples $\mu_7$, $\mu_8$, $\mu_9$ and $\mu_{10}$ (see figure 5.4).

Both pipelining and the estimation of a bound of the result of the remainder query have a huge and noticeable effect on query processing times (see figure 5.5). After the top-k semantic cache has successfully processed query $q_2$, the histogram for the estimation of query bounds has to be refreshed. Table 5.6 shows the updated tuple counts.
5. Pipelining

Figure 5.4.: Estimated bound of remainder query of query $q_2$

Figure 5.5.: Effect of pipelining of query $q_2$
5.3. Estimating Query Bounds with Histograms

Figure 5.6.: Histogram after execution of query $q_2$
5. Pipelining

Example 30 (Lower bound for remainder query $r_2$ of query $q_3$) We used the following remainder query $r_3$ for query $q_3$ (see example 19):

$$r_3 := q_3 \setminus (π_{A_3}(σ_{i_3}^r (\times_{r \in R_3} r)))$$

After removing the limit of query $q_3$, we can state the remainder condition $\hat{r}_3$ as follows:

$$\hat{r}_3 := \hat{q}_3 (\neg \hat{i}_3 \lor u\hat{i}_3)$$

The sub-expressions $\hat{q}_3$ and $\hat{i}_3$ are known (see table 4.15):

$$\hat{q}_3 = \text{created} \geq \text{last hour}$$

$$\hat{i}_3 = ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20)))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16))$$

Using the expressions $\hat{q}_3$ and $\hat{i}_3$, we can calculate the remainder condition $\hat{r}_3$ as follows:

$$\hat{r}_3 := \hat{q}_3 \land (\neg \hat{i}_3 \lor u\hat{i}_3)$$

$$\equiv (((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor ((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor u(((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\equiv (((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor ((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor (((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor u(((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor (((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor u(((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor (((\text{created} \geq \text{last hour}) \land (\neg ((\text{created} \geq \text{last hour}) \lor ((\text{created} \geq \text{today}) \land (\text{likes} \geq 20))))$$

$$\land ((\text{likes} > 160) \lor (\text{id} \leq 16)))$$

$$\lor (\text{likes} \perp) \land (\text{id} > 16)$$

$$\equiv (((\text{created} \geq \text{last hour}) \lor (\text{likes} \perp) \land (\text{id} > 16))$$

$$\lor (((\text{created} < \text{last hour}) \land (\text{likes} < 20))$$
5.3. Estimating Query Bounds with Histograms

\[ \vee ((\text{likes} \leq 160) \land (\text{id} > 16))) \]
\[ \vee ((\text{created} \geq \text{last\_hour}) \land (\text{likes} \perp) \land (\text{id} > 16)) \]
\[ \equiv ((\text{created} \geq \text{last\_hour}) \land (\text{likes} \leq 160) \land (\text{id} > 16)) \]
\[ \equiv (\text{created} \geq \text{last\_hour}) \land (\text{id} > 16) \land ((\text{likes} \leq 160) \lor (\text{likes} \perp)) \]

The remainder condition \( \tilde{r}_3 \) is visualized in the diagram of histogram \( H^e \) in figure 5.7, which shows the histogram before processing of query \( q_3 \) and before execution of the remainder query \( r_3 \).

Figure 5.7.: Histogram before execution of query \( q_3 \)
We use algorithm `getLowerBound(\(H^e, r_3, \text{server}\))` (see algorithm 14) to calculate a lower bound for the remainder query \(r_3\):

1. \(\rightarrow \text{getLowerBound}(H^e, r_3, \text{server}) \) // see algorithm 14
2. \( s := ((1, 10, \text{last week}), (28, 250, \text{now}))\)
3. \(\rightarrow \text{getSmallestBucket}(H^e, r_3, \text{server}, 1, s) \) // see algorithm 15
4. \(\rightarrow \text{exitCondition}(H^e, r_3, \text{server}, s) \) // see algorithm 16
5. \( \leftarrow (\text{false}, \perp) \)
6. \(\rightarrow \text{chooseRegions}(H^e, r_3, \text{server}, 1, s) \) // see algorithm 18
7. \(\leftarrow (1, S := [s_1 := ((1, 170, \text{last week}), (28, 250, \text{now})), s_2 := ((1, 10, \text{last week}), (28, 160, \text{now}))))\)
8. \(\rightarrow \text{processRegions}(H^e, r_3, \text{server}, 1, S) \) // see algorithm 20
9. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_1) \) // see algorithm 21
10. \(\leftarrow \text{false} \)
11. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_2) \)
12. \(\leftarrow \text{true} \)
13. \(\rightarrow \text{getSmallestBucket}(H^e, r_3, \text{server}, s_2) \)
14. \(\rightarrow \text{exitCondition}(H^e, r_3, \text{server}, s_2) \)
15. \(\leftarrow (\text{false}, \perp) \)
16. \(\rightarrow \text{chooseRegions}(H^e, r_3, \text{server}, 1, s_2) \)
17. \(\leftarrow (1, S_2 := [s_{21} := ((1, 90, \text{last week}), (28, 160, \text{now})), s_{22} := ((1, 10, \text{last week}), (28, 80, \text{now}))))\)
18. \(\rightarrow \text{processRegions}(H^e, r_3, \text{server}, 1, S_2) \)
19. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_{21}) \)
20. \(\leftarrow \text{false} \)
21. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_{22}) \)
22. \(\leftarrow \text{true} \)
23. \(\rightarrow \text{getSmallestBucket}(H^e, r_3, \text{server}, s_{21}) \)
24. \(\rightarrow \text{exitCondition}(H^e, r_3, \text{server}, s_{21}) \)
25. \(\leftarrow (\text{false}, \perp) \)
26. \(\rightarrow \text{chooseRegions}(H^e, r_3, \text{server}, 1, s_{21}) \)
27. \(\leftarrow (2, S_{21} := [s_{211} := ((1, 90, \text{last week}), (15, 160, \text{now})), s_{212} := ((16, 90, \text{last week}), (28, 160, \text{now}))))\)
28. \(\rightarrow \text{processRegions}(H^e, r_3, \text{server}, 2, S_{21}) \)
29. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_{211}) \)
30. \(\leftarrow \text{false} \)
31. \(\rightarrow \text{overlapsRegion}(\hat{r}_3, s_{212}) \)
32. \(\leftarrow \text{true} \)
33. \(\rightarrow \text{getSmallestBucket}(H^e, r_3, \text{server}, 2, s_{212}) \)
34. \(\rightarrow \text{exitCondition}(H^e, r_3, \text{server}, s_{212}) \)
At the beginning, the algorithm getLowerBound\(H^e, r_3, \text{server}\) (see algorithm [14]) initializes the considered region. Again, we call it region \(s\) instead of region \(r\) to avoid confusion with remainder query \(r_3\). Initially, the considered region \(s\) covers the whole histogram, i.e., \(s := ((1, 10, \text{last}\_\text{week}), (28, 250, \text{now}))\). The region \(s\) contains all buckets, i.e., \(b_1, \ldots, b_8\).

Thereafter, to find the smallest bucket of the histogram with respect to \text{order by} \(O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))\), the algorithm getSmallestBucket\(H^e, r_3, \text{server}, 1, s\) is called (see algorithm [15]). That algorithm checks its exit condition exitCondition\(H^e, r_3, \text{server}, s\) (see algorithm [16]). The whole histogram, which region \(s\) covers, contains more than one bucket with a positive tuple count, namely the buckets \(b_4, b_5, b_7\) and \(b_8\) (see figure [5.7]). Hence, the exit condition is not met. The region \(s\) is split into sub-regions \(S := [s_1, s_2]\) by algorithm chooseRegions\(H^e, r_3, \text{server}, 1, s\) (see algorithm [18]). Because the remainder query \(r_3\) uses the \text{order by} \(O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))\), the split utilizes the dimension \text{likes}. The sub-region \(s_1 := ((1, 170, \text{last}\_\text{week}), (28, 250, \text{now}))\) contains the buckets \(b_1, b_2\) and \(b_3\). And the sub-region \(s_2 := ((1, 10, \text{last}\_\text{week}), (28, 160, \text{now}))\) includes the buckets \(b_3, b_5, b_6, b_7\) and \(b_8\).

Algorithm processRegions\(H^e, r_3, \text{server}, 1, S\) (see algorithm [20]) processes the regions \(s_1\) and \(s_2\). It begins with the smaller region \(s_1\). But region \(s_1\) does not overlap with the remainder query \(r_3\), which is determined by algorithm overlapsRegion\(r_3, s_1\) (see algorithm [21]). Consequently, the algorithm processRegions\(H^e, r_3, \text{server}, 1, S\) processes the region \(s_2\) next. Region \(s_2\) does overlap with the remainder query \(r_3\) (see figure [5.7]). Therefore, the algorithm getSmallestBucket\(H^e, r_3, \text{server}, s_2\) is called recursively.

The algorithm getSmallestBucket\(H^e, r_3, \text{server}, s_2\) checks its exit condition. The region \(s_2\) contains more than one bucket with a positive tuple count, the buckets \(b_4, b_5, b_7\) and \(b_8\). Hence, the algorithm exitCondition\(H^e, r_3, \text{server}, s_2\) concludes that the exit condition is not met. The algorithm chooseRegions\(H^e, r_3, \text{server}, 1, s_2\) splits the region \(s_2\) into sub-regions \(S_2 := [s_{21}, s_{22}]\). Again, the split uses the dimension \text{likes},
5. Pipelining

because the remainder query $r_3$ uses the order by $O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))$. The sub-region $s_{21} := ((1, 90, \text{last week}), (28, 160, \text{now}))$ covers the buckets $b_4$ and $b_5$. The sub-region $s_{22} := ((1, 10, \text{last week}), (28, 80, \text{now}))$ contains the buckets $b_6$, $b_7$ and $b_8$. Then, the sub-regions $s_{21}$ and $s_{22}$ are processed by algorithm processRegions($H^e$, $r_3$, server, 1, $S_2$). It begins with the smaller region $s_{21}$, which overlaps with the remainder query $r_3$ (see figure [5.7]). Therefore, once again, the algorithm getSmallestBucket($H^e$, $r_3$, server, $s_{21}$) is called recursively.

The exit condition exitCondition($H^e$, $r_3$, server, $s_{21}$) of the algorithm getSmallestBucket($H^e$, $r_3$, server, $s_{21}$) is still not met, because region $s_{21}$ contains the buckets $b_4$ and $b_5$. Both of which have a positive tuple count. Consequently, the region $s_{21}$ is split into sub-regions $S_{21} := \{s_{211}, s_{212}\}$. This time, the region cannot be split any further using dimension likes. Hence, the attribute that is used in the next component of the order by $O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))$ of the remainder query $r_3$ is used, i.e., the dimension id. The sub-region $s_{211} := ((1, 90, \text{last week}), (15, 160, \text{now}))$ contains the bucket $b_4$. And the sub-region $s_{212} := ((16, 90, \text{last week}), (28, 160, \text{now}))$ contains the bucket $b_5$.

Algorithm processRegions($H^e$, $r_3$, server, 1, $S_{21}$) processes the regions $s_{211}$ and $s_{212}$. It starts with the smaller region $s_{211}$. This region does not overlap with the remainder query $r_3$. Then, it processes the region $s_{212}$. That region does overlap with the remainder query $r_3$. The algorithm getSmallestBucket($H^e$, $r_3$, server, 2, $s_{212}$) is called recursively. Finally, the exit condition exitCondition($H^e$, $r_3$, server, $s_{212}$) is met. The region $s_{212}$ contains only one bucket, the bucket $b_5$. This bucket has a positive tuple count. And it overlaps with the remainder query $r_3$. The smallest bucket has been found. It is bucket $b_5$.

The last step of algorithm getLowerBound($H^e$, $r_3$, server) is the creation of a lower bound for bucket $b_5$ based on the bounds of the bucket and the order by $O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))$ of the remainder query $r_3$ with algorithm getLowerBound($b_5$, $O_3 = ((\text{likes}, \uparrow), (\text{id}, \uparrow))$) (see algorithm [11]). That algorithm returns the tuple $\beta_3 = (10, 160, \bot, \ldots, \bot)$.

The result of algorithm getLowerBound($H^e$, $r_3$, server) is that all tuples that are transferred from the server using the remainder query $r_3$ are greater than or equal to the tuple $\beta_3 = (10, 160, \bot, \ldots, \bot)$ with respect to the order by $O_3 = ((\text{likes}, \downarrow), (\text{id}, \uparrow))$ of the remainder query $r_3$. Therefore, any tuple that is transferred from the server has less than 160 ($\beta_3[\text{likes}]$) likes or has exactly 160 ($\beta_3[\text{likes}]$) likes and an id equal to or greater than 10 ($\beta_3[\text{id}]$).

The overlap of query $q_3$ and segment $s_2$ contains the tuples $\mu_{10}$, $\mu_{19}$, $\mu_9$, $\mu_{16}$, $\mu_{13}$ and $\mu_{12}$ (see table [5,10].

The tuples $\mu_{10}$, $\mu_{19}$ and $\mu_9$ have more than 160 likes. Consequently, as the result of algorithm getLowerBound($H^e$, $r_3$, server), they can be returned immediately. They can be pipelined. The top-k semantic cache does not have to wait for the first tuple of the remainder query to return the tuples $\mu_{10}$, $\mu_{19}$ and $\mu_9$ (see figure [5.8]).
5.3. Estimating Query Bounds with Histograms

<table>
<thead>
<tr>
<th>post</th>
<th>id</th>
<th>likes</th>
<th>created</th>
<th>...</th>
<th>q₁</th>
<th>q₂</th>
<th>q₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>μ₁₀</td>
<td>10</td>
<td>230</td>
<td>last hour</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>μ₁₉</td>
<td>19</td>
<td>200</td>
<td>last hour</td>
<td>6</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>μ₉</td>
<td>9</td>
<td>170</td>
<td>last hour</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>μ₁₆</td>
<td>16</td>
<td>140</td>
<td>last hour</td>
<td>12</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>μ₁₃</td>
<td>13</td>
<td>80</td>
<td>last hour</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>μ₁₂</td>
<td>12</td>
<td>40</td>
<td>last hour</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10.: Tuples of the overlap of query q₃ and segment s₂

Figure 5.8.: Estimated bound of remainder query of query q₃
5. Pipelining

5.3.5. Generalization of Supported Orderings

The presented algorithm for the estimation of bounds of query results (see algorithm 14) currently only supports lexicographical orderings based on the attributes. But the proposed algorithm can be straightforwardly extended to orderings based on component-wise monotonous score functions.

**Definition 26 (Component-wise monotonous)** Let $A = \{a_1, \ldots, a_n\}$ be a set of attributes. Let $T = \text{dom}(a_1) \times \ldots \times \text{dom}(a_n)$ be a tuple space. And let $f : T \rightarrow \mathbb{R}$ be a score function from the tuple space $T$ to the real numbers $\mathbb{R}$ (see definition 15). The score function $f$ is called component-wise monotonous iff.

$$\forall i \in \{1, \ldots, n\} : ( \forall \mu_1, \mu_2 \in T : (\mu_1[a_1, a_{i-1}, a_{i+1}, a_n] = \mu_2[a_1, a_{i-1}, a_{i+1}, a_n] \land \mu_1[a_i] \leq \mu_2[a_i]) \Rightarrow (f(\mu_1) \leq f(\mu_2)))$$

$$\lor$$

$$\forall \mu_1, \mu_2 \in T : (\mu_1[a_1, a_{i-1}, a_{i+1}, a_n] = \mu_2[a_1, a_{i-1}, a_{i+1}, a_n] \land \mu_1[a_i] \leq \mu_2[a_i]) \Rightarrow (f(\mu_1) \geq f(\mu_2))).$$

**Example 31 (Component-wise monotonous)** Let $T_{post}$ be the tuple space of relation $post$. Consider the following score function $f$ (see example 15):

$$f : T_{post} \rightarrow \mathbb{R}, \mu \mapsto \mu[\text{likes}]^2 - \mu[\text{dislikes}]^2$$

Because the domains of both the attributes $\text{likes}$ and $\text{dislikes}$ are the natural numbers, the score function $f$ is obviously monotonically increasing in the attribute $\text{likes}$ and monotonically decreasing in the attribute $\text{dislikes}$. Hence, the score function $f$ is component-wise monotonous.

Consider a grid structured multidimensional histogram. For a query with a lexicographical ordering, there exists one single smallest bucket that can be used to compute a lower bound of the result of the given query. Algorithm getSmallestBucket($H^e, q, \tau, 1, r$) (see algorithm 15) can be used to efficiently find this smallest bucket.

For orderings using arbitrary score functions, the calculation of a lower bound using a histogram is not possible, because every point of every non-empty bucket that overlaps with the given query could be the lower bound of the result of the query. Hence, we would have to compute the score of every point of every non-empty, overlapping bucket, which is unfeasible and, for certain domains, e.g., the real numbers, actually impossible. However, if the ordering of a given query uses a score function that is component-wise monotonous, and if we know, for each component, whether the score function is monotonically increasing or monotonically decreasing, it is indeed possible to calculate a lower bound of the result of the given query. As noted above, for a query with a
5.3. Estimating Query Bounds with Histograms

Lexicographical ordering, there is one single smallest bucket. In contrast, for a query with an ordering based on a component-wise monotonous score function, there is a skyline of smallest buckets. This is illustrated by the following example.

**Example 32 (Skyline of smallest buckets)** Figure 5.9 shows the skyline of smallest buckets of a grid structured multidimensional histogram based on the attributes *likes* and *dislikes* of a query $q$ with an ordering based on function $f$ from example [31].

![Figure 5.9: Example for a skyline of buckets of a query $q$ with an ordering based on function $f$ from example [31] in a grid structured multidimensional histogram based on the attributes *likes* and *dislikes*](image)

First of all, buckets that do not overlap with the given query are irrelevant and can be ignored (noted as *irr.* in the figure). Secondly, buckets that do not contain any tuples can also be disregarded (noted as *0*). Thirdly, buckets that are greater than other non-empty, overlapping buckets with respect to the ordering, which is based on the component-wise monotonous score function $f$, do not need to be considered (noted as *gr.*). As already analyzed above in example [31], a tuple is greater than another with respect to the score function $f$ if it has more likes or less dislikes, respectively. The remaining buckets constitute a skyline of smallest bucket (noted as *sky*).

After determining the skyline, for each of the buckets of the skyline, we compute a tuple that represents the smallest point of the bucket. In the example in figure 5.9, these are the tuples $\beta_1$, $\beta_2$, and $\beta_3$. The lower bound $\beta$ of the result of the given query $q$ is the minimum of these tuples with respect to the ordering of the query, i.e., $\beta := \min_{\mathcal{O}}\{\beta_1, \beta_2, \beta_3\}$. 
Algorithm getLowerBound($H^e$, $q$, $\tau$) (see algorithm 23) shows how the skyline operator can be incorporated into the proposed algorithm getLowerBound($H^e$, $q$, $\tau$) of the previous sections (see algorithm 10 and algorithm 14). To efficiently implement this extension of the algorithm, we need to be able to quickly identify the skyline of smallest bucket with respect to a given query, which uses a certain component-wise monotonous ordering.

Algorithm 23 getLowerBound($H^e$, $q$, $\tau$) — for orderings based on component-wise monotonous functions

Input: histogram for the estimation of query bounds $H^e = (A^{grid}, A^{all}, R, B, I)$, query $q = (\ldots, \hat{q}, O, \ldots)$ and tuple count $\tau$

Output: lower bound $\beta$

1. $\beta := \min_{O} \{ \mu \mid \\
   \exists b_{skyline} : ( \\
   \mu = \text{getLowerBound}(b_{skyline}, O) \\
   \wedge b_{skyline} \in \text{skyline}_{O}( \\
   \{ b \mid b \in B \\
   \wedge b = (l, u, t) \\
   \wedge t(\tau) > 0 \\
   \wedge \text{overlapsBucket}(\hat{q}, b) \\
   \}) \\
   ) \}$

2. return $\beta$

Fortunately, skyline algorithms have received much interest. Many algorithms for skyline computation have been proposed: divide and conquer (D&C) [Mat91, BKS01], block nested loop (BNL) [BKS01], bitmap [TE01], index [TE01], neared neighbor (NN) [KRR02], sort filter skyline (SFS) [CGGL03, CGGL05], branch and bound skyline (BBS) [PTFS03, PTFS05], linear elimination sort for skyline (LESS) [GSG05], the sort and limit skyline algorithm (SaLSa) [BCP08], object-based space partitioning (OSPS) [ZMC09], Z-order curve (Z-Sky) [LLZ10], balanced pivot point selection (BSkyTree) [LH10], and sort first skyline join (SFSJ) [VDP11].

Furthermore, our algorithm for the estimation of query bounds can also be extended to certain score functions that are not component-wise monotonous. If the score function is constructed using arbitrary attribute score functions (see definition 13) and a component-wise monotonous combining function (see definition 14), we can create a multidimensional histogram that is built on the results of the attribute score functions instead of the values of the attributes themselves. Because the combining function is component-wise monotonous, the algorithm for the estimation of query bounds can...
operate as described above, i.e., find the skyline of smallest buckets, and compute a lower bound of the result of a given query. However, the top-k semantic cache needs to know the combinations of utilized attribute score functions beforehand for this approach to work, because it has to prepare and maintain suitable multidimensional histograms.

For future work, we plan to explore the adaption of known, efficient skyline algorithms to our needs and their integration into our innovative algorithm for lower bound estimation to enable in the top-k semantic cache the pipelining of queries with orderings that use component-wise monotonous score functions or attribute score functions in combination with component-wise monotonous combining functions, respectively.
6. Top-k Semantic Caching

This chapter discusses query processing in a top-k semantic cache. First, we discuss cache organization. Then, we examine in much detail all aspects of query processing: how to find overlapping segments, how to load missing attributes, how to load more tuples, and, most importantly, how to answer a query. In addition, we explain the different coalescing strategies.

In this chapter, we use the introduced definitions and operations on segments (see chapter 4). And, to enable pipelining in the top-k semantic cache, we apply the proposed algorithm for the estimation of query bounds (see chapter 5).

6.1. Organization

When a query enters the top-k semantic cache, it is parsed, analyzed and described by meta data. (Note that meta data is cached as well using a separate meta data cache.) Therefore, the top-k semantic cache is capable of zero level rejection [AQS08]. That means, it is able to directly reject queries with incorrect attributes (e.g., that are not contained in any referenced table), invalid table names (i.e., that are not in the database) or invalid conditions (e.g., a comparison between an integer and a varchar).

If the projection \( A_{\text{select}} \) of the query \( q \) does not contain the primary attributes of the referenced tables, the top-k semantic cache will add them. Similarly, if the projection of the query does not contain all attributes that are used in the order by of the query, the top-k semantic cache will add them to the projection as well to satisfy the definition of a segment (see definition 9).

The segments of the top-k semantic cache are divided into equivalence classes \( \text{Cache}_{R,J} \). Two segments are in the same equivalence class if they have the same set of tables \( R \) and if these tables are equi-joined the same way using the equi-joins \( J \). The referenced tables can be read from the from clause, because we only consider sorted and limited projection-selection-join queries. The equi-joins can be efficiently determined from the where condition in linear time with respect to the length of the where condition by algorithm getJoins(\( q \)) (see algorithm 24). This logical partitioning of the top-k semantic cache provides faster query processing. In addition, it allows for better prediction of the lower bound for the first tuple from server.
6. Top-k Semantic Caching

Example 33 (Equivalence classes in running example) The running example consists of the five tables author, post, postcategory, image, and imagecategory. Figure 6.1 shows the graph of foreign key constraints of the tables of the running example (see section 1.2).

![Figure 6.1: Graph of the foreign key constraints of the tables of the running example](image)

There are five tables, four possible joins for two tables and four possible joins for three tables (see figure 6.1).

<table>
<thead>
<tr>
<th>two tables</th>
<th>three tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>author ← post</td>
<td>author ← post ← postcategory</td>
</tr>
<tr>
<td>post ← postcategory</td>
<td>author ← post ← post ← image</td>
</tr>
<tr>
<td>post ← image</td>
<td>image ← post ← postcategory</td>
</tr>
<tr>
<td>image ← imagecategory</td>
<td>post ← image ← imagecategory</td>
</tr>
</tbody>
</table>

Table 6.1: Possible joins of running example with two and three tables

The algorithm getJoins(q) (see algorithm 24) returns all joins of the query q that can be directly inferred by its where condition ˆq. First, it calculates the set of all unordered pairs of attributes that are equated within the where condition ˆq. Then it iterates all possible joins of relations R of query q. If all attributes of the considered join that should be equated are contained in the set of actually equated attributes of the where condition ˆq, the join is added to the set J of joins of the query q.

The recursive algorithm getEqualAttributes(ˆe) (see algorithm 25) is used to calculate the set of all unordered pairs of attributes that are equated within the expression ˆe. If the expression ˆe is a conjunction, the equation of two attributes must hold in one sub-expression. On the other hand, if the expression ˆe is a disjunction, the equation of two attributes must hold in every sub-expression.
Algorithm 24 getJoins(q)

**Input:** query q  
**Output:** set of Joins J

1: $J := \emptyset$
2: $E := \text{getEqualAttributes}(\hat{q})$  
   // see algorithm 25
3: for each $j$ is possible join of relations in $R$ do
4: $e := \text{equated attributes of join } j$
5: if $e \subseteq E$ then
6: $J := J \cup j$
7: end if
8: end for
9: return $J$

Algorithm 25 getEqualAttributes(\hat{c})

**Input:** expression $\hat{c}$  
**Output:** set of equal attributes $E$

1: if $\hat{c} = (x = y)$ with $x$, $y$ attributes then
2: return $\{ \{x, y\}\}$
3: else if $\hat{c} = \bigwedge_{i=1}^{n} \hat{c}_i$ then
4: return $\bigcup_{i=1}^{n} \text{getEqualAttributes}(\hat{c}_i)$
5: else if $\hat{c} = \bigvee_{i=1}^{n} \hat{c}_i$ then
6: return $\bigcap_{i=1}^{n} \text{getEqualAttributes}(\hat{c}_i)$
7: else
8: return $\{\}$
9: end if
6. Top-k Semantic Caching

Example 34 (Joins of a query) Consider the following query $q_j$:

```sql
SELECT * FROM post p, image i, imagecategory ic
WHERE (p.id = i.post AND ic.image = i.id AND ic.post = i.post AND ic.name = 'Cats')
OR (p.id = i.post AND ic.image = i.id AND ic.post = i.post AND ic.name = 'Dogs')
ORDER BY p.likes DESC
LIMIT 10
```

Listing 6.1: Query $q_j$: a top-k query with joins

Obviously, the query $q_j$ uses the relations `post`, `image` and `imagecategory`:

$$R = \{\text{post, image, imagecategory}\}$$

The set of unordered pairs of equated attributes is calculated by algorithm `getEqualAttributes(\tilde{q}_j)` (see algorithm 25) as follows:

$$E_j := \text{getEqualAttributes}(\tilde{q}_j)$$

$$= \text{getEqualAttributes}(\text{where } (p.id = i.post \land ic.image = i.id \land ic.post = i.post \land ic.name = 'Cats')$$

$$\lor (p.id = i.post \land ic.image = i.id \land ic.post = i.post \land ic.name = 'Dogs'))$$

$$= \text{getEqualAttributes}(\text{where } (p.id = i.post \land ic.image = i.id \land ic.post = i.post \land ic.name = 'Cats'))$$

$$\cap \text{getEqualAttributes}(\text{where } (p.id = i.post \land ic.image = i.id \land ic.post = i.post \land ic.name = 'Dogs'))$$

$$= \{(p.id, i.post), \{ic.image, i.id\}, \{ic.post, i.post\}\}$$

$$\cap \{(p.id, i.post), \{ic.image, i.id\}, \{ic.post, i.post\}\}$$

$$= \{(p.id, i.post), \{ic.image, i.id\}, \{ic.post, i.post\}\}$$

The join $p \xleftarrow{\text{post}} i$ of the relations `post` and `image` is supported by $E_j$, because $\{(p.id, i.post)\} \subseteq E_j$ holds. Similarly, the join $i \xleftarrow{\text{image}} ic$ of the relations `image` and `imagecategory` is supported by $E_j$, because $\{(i.id, ic.image), \{i.post, ic.post\}\} \subseteq E_j$ holds.

Hence, $q_j \in \text{Cache}_{R,J}$ with $R = \{\text{post, image, imagecategory}\}$ and $J = \{\text{post} \xleftarrow{\text{image}} \text{image, image} \xleftarrow{\text{post}} \text{image, imagecategory}\}$. 
6.2. Query Processing

In this section, we examine the query processing of a top-k semantic cache. First, we briefly show how the segments that overlap with a query can be found. Afterward, we explain how missing attributes can be retroactively loaded. Thereafter, we discuss how the top-k semantic cache answers a query. Finally, we talk about the different coalescing strategies for cache reorganization and their implementation.

6.2.1. Find Overlapping Segments

Algorithm cacheOverlappingSegments(Cache_{R,J}, q) (see algorithm 26) returns all overlapping segments $L$ of the equivalence class Cache_{R,J} of a query $q$.

Algorithm 26 cacheOverlappingSegments(Cache_{R,J}, q)

Input: equivalence class Cache_{R,J} of the top-k semantic cache, query $q$

Output: overlapping segments $L$

1: $L := \emptyset$
2: for each $s_c \in$ Cache_{R,J} do
3: if Solver.overlaps($\hat{s}_c, \hat{q}$) then
4: $L := L \cup \{s_c\}$
5: end if
6: end for
7: return $L$

The algorithm just iterates over all segments in the same equivalence class Cache_{R,J} as query $q$. It uses an SMT solver to detect overlaps. The algorithm is used in example 17 and example 19.

6.2.2. Load Missing Attributes

By definition (see definition 9), a segment must always contain the primary attributes of all tables of the segment. Therefore, only non-primary attributes can be missing to answer a query. And because all primary attributes are present in the segment, the algorithm is able to load missing non-primary attributes. A segment that misses non-primary attributes to answer a query is called vertically partitioned [RDK03]. Algorithm segmentLoadAttributes($s, A^{new}$) (see algorithm 27) loads missing non-primary attributes $A^{new}$ for a segment $s$.

According to proposition 10, the loaded tuples of a segment are described by the condition $(\hat{g} \leq \lambda \lor \hat{i}) \land \hat{s}$. Hence, the following query loads the values of the new attributes $A^{new}$:

$$
\pi_{primary(A^{select})} \cup A^{new}(\sigma_{(\hat{g} \leq \lambda \lor \hat{i}) \land \hat{s}}(\times r \in R^r))
$$
6. Top-k Semantic Caching

**Algorithm 27** segmentLoadAttributes($s$, $A_{new}$)

**Input:** $s$, $A_{new}$
- with $A_{new} \subseteq A_{from}$
- and $A_{new} \cap A_{select} = \emptyset$

**Output:** $s$

1. $T := T \Join \pi_{primary}(A_{select}) \cup A_{new} (\sigma_{(\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)} \times \rho_{r \in R} r)$
2. $A_{select} := A_{select} \cup A_{new}$
3. $\{p_1, \ldots, p_n\} := primary(A_{select})$
4. $\lambda := \sigma_{i=1}^n (p_i = \lambda[p_i]) (T)$
5. return $s$

Such a query that loads missing non-primary attributes is called an amending query [RDK03]. To be able to insert the new information into the cache, the primary attributes must also be selected. Doing so, the insertion of the new information can be done by a natural join. Finally, the values of the new attributes have to be set for the last loaded tuple $\lambda$.

**Example 35 (Loading of additional columns for segment $s_1$)** Consider the segment $s_1$ after subtraction of segment $s_2$. The expressions $\hat{s}_1$, $\hat{i}_1$ and $\hat{o}_1^{\leq \lambda}$ are defined as follows (see Table 4.12 and example 13):

$\hat{s}_1 = (\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100)$

$\hat{i}_1 = \text{false}$

$\hat{o}_1^{\leq \lambda} = (\text{likes} \leq \perp) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2))$

Using these expressions, we can calculate $(\hat{o}_1^{\leq \lambda} \lor \hat{i}_1) \land \hat{s}_1$ as follows:

$((\hat{o}_1^{\leq \lambda} \lor \hat{i}_1) \land \hat{s}_1) = ((\text{likes} \leq \perp) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$

$\lor (\text{false})$

$\land (((\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100))$

$\equiv ((\text{likes} \leq \perp) \lor (\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$

$\land ((\text{created} \geq \text{last week}) \land (\text{created} < \text{today}) \land (\text{likes} \geq 100))$

$\equivtrue ((\text{likes} > 160) \lor ((\text{likes} = 160) \land (\text{id} \leq 2)))$

$\land (\text{created} \geq \text{last week}) \land (\text{created} < \text{today})$.

Hence, following SQL query $q_{content}$ loads the missing attribute content for all tuples of segment $s_1$:

$\text{SELECT p.id, p.content}$
6.2. Query Processing

```sql
FROM post p
WHERE ((p.likes > 160) OR ((p.likes = 160) AND (p.id <= 2)))
AND (p.created >= LAST_WEEK) AND (p.created < TODAY)
```

Listing 6.2: Query that loads the missing attribute content

6.2.3. Load More Tuples

Sometimes, the top-k semantic cache encounters a query \( q \) that he encountered before – possibly with a different limit. And the cache contains a segment \( s_c \) with \( \hat{q} \equiv \hat{s_c}, O_q = O_c \) and \( A^q_{\text{select}} \subseteq A^c_{\text{select}} \). Then, it can use this segment \( s_c \) to directly answer the query \( q \). But if \( k_q > k_c \) holds, the segment \( s_c \) will need more tuples to answer query \( q \).

The needed tuples can be loaded by algorithm segmentLoadTuples\((s, l)\) (see algorithm 28).

**Algorithm 28** segmentLoadTuples\((s, l)\)

**Input:** segment \( s \), number of tuples \( l \) to load

**Output:** modified segment \( s \)

1. \( N := \text{limit}(\text{orderby}_{O_c}(\pi_A_{\text{select}}(\sigma(\hat{c} \land \neg \hat{u} \lor \hat{u}))) \times r_{c})) \)
2. \( T := T \cup N \)
3. \( \lambda := \max_{O_c}(N) \)
4. \( k := k + |N| \)
5. return \( s \)

According to proposition [T1], the tuples that are contained by segment \( s_c \) but have not yet been loaded, are described by the expression \( \hat{c} \land (\neg \hat{u} \lor \hat{u}) \land \hat{s_c} \)\).

Hence, we can load up to \( l \) more tuples into segment \( s_c \) using the query

\[
\text{limit}(\text{orderby}_{O_c}(\pi_{A_{\text{select}}}(\sigma(\hat{c} \land \neg \hat{u} \lor \hat{u}))) \times r_{c}))
\]

6.2.4. Answer Query

The top-k semantic cache answers queries in a highly parallelized way. It essentially performs a parallelized merge sort (see figure 6.2). It uses mapper threads and reducer threads similar to the MapReduce programming model [DG08].

**Algorithm cacheAnswerQuery(Cache, q)** (see algorithm 29) describes how the top-k semantic cache \( Cache \) answers a query \( q \). Note that the all concurrency has been removed from the algorithm for easier understanding.

First of all, the algorithm determines the equi-joins \( J \) that are used in the query \( q \). Together with the tables \( R \) of the query, the equi-joins are used to choose the correct
6. Top-k Semantic Caching

Figure 6.2.: A top-k semantic cache answers a query
6.2. Query Processing

Algorithm 29 cacheAnswerQuery(Cache, q)

Input: top-k semantic cache Cache, query q

Output: result T of query q

1: \( J := \text{getJoins}(q) \) \hspace{1cm} // see algorithm 24
2: Cache\(_{R,J} := \text{Cache}(R, J) \)
3: \( L := \text{cacheOverlappingSegments}(\text{Cache}_{R,J}, q) \) \hspace{1cm} // see algorithm 26
4: for each \( s_c \in L \) do
5: \( A^\text{select}_c := A^\text{select} \cup A^\text{where} \setminus A^\text{select} \)
6: \( \text{segmentLoadAttributes}(s_c, A^\text{missing}_c) \) \hspace{1cm} // see algorithm 27
7: end for
8: if \( L \neq \emptyset \) then
9: \( \hat{q} := \bigvee_{s_c \in L} (\widehat{c} \lor \widehat{\alpha} \land \widehat{c}) \) \hspace{1cm} // see proposition 10
10: if \( \neg \text{Solver}.\text{isSatisfiable}((\neg \hat{q} \lor \text{u} \hat{q}) \land T) \) then
11: \( T := \text{limit}_k(\text{orderby}_O(\bigcup_{q_c \in L}(\pi_{A^\text{select}}(\sigma_{\hat{q}}(T_c)))))) \)
12: return \( T \)
13: end if
14: \( \alpha := \text{getLowerBound}(H', q, \text{server}) \) \hspace{1cm} // see algorithm 14
15: \( T_{\leq \alpha} := \text{limit}_k(\text{orderby}_O(\bigcup_{q_c \in L}(\pi_{A^\text{select}}(\sigma_{\hat{q}}(T_c)))))) \)
16: if \( |T_{\leq \alpha}| = k \) then
17: \( T := T_{\leq \alpha} \)
18: return \( T \)
19: end if
20: \( T_r := q \setminus \pi_{A^\text{select}}(\sigma_{\hat{x}}(x \in R')) \) \hspace{1cm} remainder query \( r \)
21: \( T_{> \alpha} := \text{limit}_{k-|T_{\leq \alpha}|}(\text{orderby}_O(\bigcup_{q_c \in L}(\pi_{A^\text{select}}(\sigma_{\hat{q}}(\geq \alpha}(T_c)))))) \)
22: \( T := T_{\leq \alpha} \cup T_r \cup T_{> \alpha} \) \hspace{1cm} // \( (L = \emptyset) \)
23: return \( T \)
24: end if
25: if shouldAddSegment(q) then
26: \( \text{cacheAddFullCoal}(\text{Cache}_{R,J}, q, L, T_r, l) \) \hspace{1cm} // see algorithm 30
27: \( \text{cacheAddNoCoal}(\text{Cache}_{R,J}, q, L, T_r, l) \) \hspace{1cm} // see algorithm 31
28: return \( T \)
29: end if
30: return \( T \)
6. Top-k Semantic Caching

equivalence class \( Cache_{R,J} \) of segments of the cache. The algorithm \( cacheOverlappingSegments(Cache_{R,J}, q) \) (see algorithm \[26\]) is used to find the segments \( L \) that overlap with the query \( q \).

Each overlapping segment must contain the attributes \( A^{select} \cup A^{where} \). It must contain \( A^{select} \), because that are the attributes that are projected by the query. And it must contain \( A^{where} \), because otherwise the tuples that are selected by the query could not be identified. It must also contain all attributes \( A^{orderby} \) of the \( order by \). But, by definition of a segment (see definition \[9\]), the attributes of the \( order by \) are always contained in the projection \( A^{select} \), i.e., \( A^{orderby} \subseteq A^{select} \). Therefore, the missing attributes of an overlapping segment \( s_c \) are as follows:

\[
A^{missing}_c := (A^{select} \cup A^{where}) \setminus A^{select}_c
\]

They are loaded by algorithm \( segmentLoadAttributes(s_c, A^{missing}_c) \) (see algorithm \[27\]).

The following condition \( \widehat{i} \) covers all tuples that are contained in the overlaps of query \( q \) and the overlapping segments of the cache (see proposition \[10\]):

\[
\widehat{i} := \bigvee_{s_c \in L} ((\widehat{i}_c \lor \widehat{o}_c^{\leq \lambda}) \land \widehat{s}_c)
\]

Hence, according to proposition \[11\], the following remainder condition \( \widehat{r} \) describes all tuples that must be transferred from the server:

\[
\widehat{r} := (\neg \widehat{i} \lor \widehat{u}_i) \land \widehat{q}
\]

If this remainder condition \( \widehat{r} \) is not satisfiable, the top-k semantic cache does not need to contact the server to answer the query, because the query result is completely contained in the cache. The algorithm just gathers all tuples that satisfy the query \( q \) and applies the ordering \( O \) and limit \( k \):

\[
T := \text{limit}_k(orderby_O(\bigcup_{q_c \in L} (\pi_{A^{select}}(\sigma_{\widehat{q} \land \widehat{o}^{\leq \lambda} \land \widehat{s}_c}(T_c))))))
\]

On the other hand, if the remainder condition \( \widehat{r} \) is satisfiable, the algorithm has to estimate a lower bound \( \alpha \) for the first tuple from server by algorithm \( getLowerBound(H^c, q, server) \) (see algorithm \[14\]). All tuples that are smaller than or equal to the estimated lower bound \( \alpha \) can be pipelined, i.e., immediately be returned by the cached.

The condition \( \neg \widehat{o}^{\leq \alpha} \) describes all tuples that are smaller than or equal to the estimated lower bound \( \alpha \) (see definition \[10\]). The condition \( \widehat{q} \land \neg \widehat{o}^{\leq \alpha} \) describes the tuples that satisfy the query condition \( \widehat{q} \) and are smaller than or equal to \( \alpha \). Therefore, the algorithm retrieves the following set \( T_{\leq \alpha} \) of tuples:

\[
T_{\leq \alpha} := \text{limit}_k(orderby_O(\bigcup_{q_c \in L} (\pi_{A^{select}}(\sigma_{\widehat{q} \land \neg \widehat{o}^{\leq \alpha} \land \widehat{s}_c}(T_c))))))
\]
And it immediately starts to return them. If the top \( k \) tuples of the query according to the ordering \( O \) are smaller than or equal to \( \alpha \), no tuples from the server will be needed to answer the query.

Note that the condition \( \hat{o} \leq \alpha \) will not actually have to be executed on the segments. Instead, all tuples from the segments that match the query condition \( \hat{q} \) are concurrently retrieved and combined (see figure 6.2). If the lower bound \( \alpha \) for the first tuple from the server is reached, the merging has to stop until the first actual tuple from the server has been transferred to the client. Then, the algorithm can continue.

The remaining tuples that are needed to answer the query and that cannot be provided by the top-\( k \) semantic cache are retrieved using the remainder query \( r \). Since the expression \( \hat{i} \) describes all tuples that are contained in the cache and contribute to the query result of query \( q \), the remaining tuples can be loaded as follows:

\[
T_r := q \setminus \pi_A\sigma_i(\times_{r \in R} r)
\]

By construction, the remainder query \( r \) does not load any tuples that are not necessary to answer the query \( q \).

In addition, the processing overhead at the server that has to evaluate the remainder query \( r \) instead of query \( q \) is negligibly small. The database system at the server can just calculate the result of query \( q \) as before. Then, it filters the result and only returns the tuples to the client that does not satisfy the condition \( \hat{i} \). The filtering can be pipelined.

In addition, the tuples that are greater than the lower bound \( \alpha \) have to be retrieved:

\[
T_{>\alpha} := \text{limit}_{k-|T_{\leq\alpha}|-|T_r|} (\text{orderby}_O (\bigcup_{q_c \in L} (\pi_A\sigma_{\hat{i} \land \hat{o} > \alpha} (T_{c}))))
\]

Finally, the algorithm has to decide whether to add a segment for the query \( q \) to the cache and which coalescing strategy to use. Currently, it has to choose between full coalescing (see algorithm 30) and no coalescing (see algorithm 31).

The processing of query \( q_1 \) of the running example (see section 1.2) is described by the example 10 (description of query \( q_1 \)) and example 11 (creation of cache segment for query \( q_1 \)).

The processing of query \( q_2 \) of the running example is discussed in example 17 (creation of cache segment \( s_2 \) for query \( q_2 \), insertion of initially loaded tuples into segment \( s_2 \), and execution of remainder query \( r_2 \)) and in example 18 (subtraction of segment \( s_2 \) from segment \( s_1 \)).

Lastly, the processing of query \( q_3 \) of the running example is explained in example 19 (creation of cache segment \( s_3 \) for query \( q_3 \), insertion of initially loaded tuples into segment \( s_3 \), and execution of remainder query \( r_3 \)) and in example 20 (subtraction of segment \( s_3 \) from segment \( s_2 \)).
6. Top-k Semantic Caching

6.2.5. Coalescing Strategies

Consider a new query, which the top-k semantic cache currently computes the answer to. Assume that this query overlaps with one or more cached segments. An important property of the top-k semantic cache is that all cached segments must be disjoint. Therefore, it must take action to restore this disjointness property.

As introduced in section 4.2, the top-k semantic cache has several options: full coalescing, no coalescing and total union, or a combination of these approaches [JAT06]. The cache management should decide on a query-by-query basis using a cost analysis.

We will now have a closer look at the two main coalescing strategies, full coalescing and no coalescing.

6.2.5.1. Full coalescing

The algorithm cacheAddFullCoal(Cache \(R,J,q,L,T_r,l\)) (see algorithm 30) adds a segment to the top-k semantic cache using full coalescing.

Algorithm 30 cacheAddFullCoal(Cache \(R,J,q,L,T_r,l\))

**Input:** equivalence class Cache \(R,J\) of the top-k semantic cache, query \(q\), overlapping segments \(L\), query result \(T_r\) of the remainder query \(r\), total number of tuples \(l\) of the new segment

**Output:** new segment \(s_{new}\)

1: \(s_{new} := \text{segmentCreate}(q)\) // see algorithm 3
2: \(\text{segmentInitialInsert}(s_{new}, L)\) // see algorithm 4
3: \(T_{new} := T_{new} \cup T_r\)
4: \(\lambda_{new} := \max_\Omega(T_r)\)
5: if \(l < k\) then
6: \(\text{segmentComplete}(s_{new})\) // see algorithm 5
7: end if
8: Cache\(_{R,J} := \text{Cache}_{R,J} \cup \{s_{new}\}\)
9: for each \(s_c \in L\) do
10: \(s_c := s_c \setminus s_{new}\) // see algorithm 6
11: if \(k_c = 0\) then
12: Cache\(_{R,J} := \text{Cache}_{R,J} \setminus \{s_c\}\)
13: end if
14: end for
15: return \(s_{new}\)

First, the algorithm creates an empty new segment \(s_{new}\) for the query \(q\) using segmentCreate\((q)\) (see algorithm 3). Secondly, all tuples from the overlapping segments \(L\) are moved into the new segment \(s_{new}\) using segmentInitialInsert\((s_{new}, L)\) (see algorithm 4).
Thereafter, the tuples $T_r$ that have been transferred from server are inserted into the segment $s_{new}$. And the last loaded tuple $\lambda$ is updated. If the query result is smaller than the limit $k$ of the query, the query has been completely loaded. The algorithm can mark the segment as complete with algorithm $\text{segmentComplete}(s_{new})$ (see algorithm 5). Finally, the segment $s_{new}$ is added to the semantic cache. In addition, it is subtracted from all overlapping segments by calculating the in-place difference of the overlapping segment $s_c$ and the new segment $s_{new}$ (see algorithm 6). Segments that are empty after the segment $s_{new}$ has been subtracted are deleted.

**Example 36 (Full coalescing of segment $s_2$)** Full coalescing of segment $s_2$ is conducted by algorithm $\text{cacheAddFullCoal}(\text{Cache}_{\{\text{post}\}, \emptyset, q_2, \{s_1\}, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}, 10)$ (see algorithm 30) as follows:

1. $\to \text{cacheAddFullCoal}(\text{Cache}_{\{\text{post}\}, \emptyset, q_2, \{s_1\}, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}, 10)$
2. $\to \text{segmentCreate}(q_2)$ // see example 17
3. $\leftarrow s_2$
4. $\to \text{segmentInitialInsert}(s_2, \{s_1\})$ // see example 17
5. $\leftarrow s_2$
6. $T_2 := T_2 \cup \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}$
7. $\lambda_2 := \mu_{16}$ // see example 17
8. $\text{Cache}_{\{\text{post}\}, \emptyset} := \text{Cache}_{\{\text{post}\}, \emptyset} \cup \{s_2\}$
9. $s_1 := s_1 \setminus s_2$ // see example 18
10. $\leftarrow s_2$

The algorithm $\text{cacheAddFullCoal}(\text{Cache}_{\{\text{post}\}, \emptyset, q_2, \{s_1\}, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}, 10)$ (see algorithm 30) creates an empty segment $s_2$ for query $q_2$ with algorithm $\text{segmentCreate}(q_2)$ (see example 17). Afterward, it copies all tuples that are contained in the overlap with overlapping segment $s_1$ into segment $s_2$ using algorithm $\text{segmentInitialInsert}(s_2, \{s_1\})$ (see example 17). Then, the algorithm adds all tuples that have been transferred from the server by the remainder query to segment $s_2$. And it updates the last loaded tuple $\lambda_2$ accordingly (see example 17). The completed segment $s_2$ is added to the top-k semantic cache. Finally, segment $s_2$ is subtracted from the overlapping segment $s_1$ (see example 18).

**Example 37 (Full coalescing of segment $s_3$)** Full coalescing of segment $s_3$ is performed by algorithm $\text{cacheAddFullCoal}(\text{Cache}_{\{\text{post}\}, \emptyset, q_3, \{s_2\}, \{\mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\}, 10)$ (see algorithm 30) as follows:

1. $\to \text{cacheAddFullCoal}(\text{Cache}_{\{\text{post}\}, \emptyset, q_3, \{s_2\}, \{\mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\}, 10)$
2. $\to \text{segmentCreate}(q_3)$ // see example 19
3. $\leftarrow s_3$

161
6. Top-k Semantic Caching

\[ \text{segmentInitialInsert}(s_3, \{s_2\}) \] // see example 19
\[ \leftarrow s_3 \]
\[ T_3 := T_3 \cup \{\mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\} \]
\[ \lambda_3 := \mu_{27} \] // see example 19
\[ \text{Cache}_{\{\text{post}\}, \emptyset} := \text{Cache}_{\{\text{post}\}, \emptyset} \cup \{s_3\} \]
\[ s_2 := s_2 \setminus s_3 \] // see example 20
\[ \leftarrow s_3 \]

The algorithm cacheAddFullCoal(\text{Cache}_{\{\text{post}\}, \emptyset}, q_3, \{s_2\}, \{\mu_{21}, \mu_{22}, \mu_{23}, \mu_{25}, \mu_{27}\}, 10) \] (see algorithm 30) begins by creating an empty segment \( s_3 \) for query \( q_3 \) with algorithm segmentCreate(\( q_3 \)) (see example 19). Secondly, it calls algorithm segmentInitialInsert(\( s_3, \{s_2\} \)) (see example 19), which copies all tuples that are contained in the overlap with overlapping segment \( s_2 \) into segment \( s_3 \). Thirdly, the algorithm inserts all tuples that have been loaded by the remainder query into segment \( s_3 \). Then, it updates the last loaded tuple \( \lambda_3 \) accordingly (see example 19). The completed segment \( s_3 \) is added to the top-k semantic cache. In the end, the algorithm subtracts segment \( s_3 \) from the overlapping segment \( s_2 \) (see example 20).

6.2.5.2. No coalescing

The algorithm cacheAddNoCoal(\text{Cache}_{R,J}, q, L, T_r, l) \] (see algorithm 30) adds a segment to the top-k semantic cache using \textit{no coalescing}.

The algorithm starts by creating an empty segment \( s_{\text{new}} \) for the remainder of query \( q \), i.e., for the part of query \( q \) that does not overlap with any segment of the cache, which is described by \( \hat{q} \wedge (\bigwedge_{s_c \in L} (\neg \hat{s}_c \lor u\hat{s}_c)) \) (see proposition 11). All tuples that have been loaded by the remainder query are moved into the newly created segment \( s_{\text{new}} \). And the last loaded tuple \( \lambda_{\text{new}} \) of segment \( s_{\text{new}} \) is set accordingly. If the number of tuples \( l \) that have been loaded is smaller than the limit \( k \) of the query, the query has been completely loaded. Then, the algorithm marks the segment as complete with algorithm segmentComplete(\( s_{\text{new}} \)) (see algorithm 5). Afterward, segment \( s_{\text{new}} \) is added to the top-k semantic cache. No tuples from overlapping segments have to be copied into segment \( s_{\text{new}} \), because, by construction, segment \( s_{\text{new}} \) does not overlap with any segment of the cache.

For all overlaps of the query \( q \) and segments of the top-k semantic cache \( s_c \in L \), a separate segment \( s_i := q \cap s_c \) is created (see algorithm 7). If the query has been completely loaded, the new segment \( s_i \) is marked as complete using algorithm segmentComplete(\( s_i \)) (see algorithm 5). If the segment \( s_i \) is not empty, it will be added to the cache.

Finally, the query \( q \) is subtracted from all overlapping segments by calculating the in-place difference of the overlapping segment \( s_c \) and the query \( q \) (see algorithm 6). Segments that are empty after the query \( q \) has been subtracted are deleted.
Algorithm 31 cacheAddNoCoal\((Cache_{R,J}, q, L, T_r, l)\)

**Input:** equivalence class \(Cache_{R,J}\) of the top-k semantic cache, query \(q\), overlapping segments \(L\), query result \(T_r\) of the remainder query \(r\), total number of tuples \(l\) of the new segment

**Output:** new segment \(s_{\text{new}}\), new segments for all intersections \(I = \{s_1, \ldots, s_n\}\) of query \(q\) and existing segments

1. \(s_{\text{new}} := \text{segmentCreate}\((\{A \text{select}, R, \hat{q} \land (\bigwedge_{s_c \in L} (\neg \hat{s}_c \lor u\hat{s}_c))\}, O, |T_r|)\) \quad // see algorithm 3
2. \(T_{\text{new}} := T_r\)
3. \(\lambda_{\text{new}} := \max_O(T_r)\)
4. if \(l < k\) then
5. \(\text{segmentComplete}(s_{\text{new}})\) \quad // see algorithm 5
6. end if
7. \(Cache_{R,J} := Cache_{R,J} \cup \{s_{\text{new}}\}\)
8. \(I := \emptyset\)
9. for each \(s_c \in L\) do
10. \(s_i := s_c \cap q\) \quad // see algorithm 7
11. if \(k_i > 0\) then
12. if \(l < k\) then
13. \(\text{segmentComplete}(s_i)\) \quad // see algorithm 5
14. end if
15. \(I := I \cup \{s_i\}\)
16. \(Cache_{R,J} := Cache_{R,J} \cup \{s_i\}\)
17. end if
18. \(s_c := s_c \setminus q\) \quad // see algorithm 6
19. if \(k_c = 0\) then
20. \(Cache_{R,J} := Cache_{R,J} \setminus \{s_c\}\)
21. end if
22. end for
23. return \(s_{\text{new}}, I\)
6. Top-k Semantic Caching

Example 38 (No coalescing of segment $s_2$) No coalescing of segment $s_2$ is conducted by algorithm cacheAddNoCoal($\text{Cache}_{\{\text{post}\}, \emptyset}, q_2, \{s_1\}, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}, 10$) (see algorithm 30) as follows:

1. $\rightarrow \text{cacheAddNoCoal}(\text{Cache}_{\{\text{post}\}, \emptyset}, \emptyset, q_2, \{s_1\}, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}, 10)$
2. $\rightarrow \text{segmentCreate}(\{A_2^{\text{select}}, R_2, \hat{q}_2 \land (\neg \hat{s}_1 \lor u\hat{s}_1), O_2, \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}\})$
3. $\leftarrow s_2$
4. $T_2 := \{\mu_{11}, \mu_{12}, \mu_{13}, \mu_{15}, \mu_{16}\}$
5. $\lambda_2 := \mu_{16}$ // see example 17
6. $\text{Cache}_{\{\text{post}\}, \emptyset} := \text{Cache}_{\{\text{post}\}, \emptyset} \cup \{s_2\}$
7. $I := \emptyset$
8. $s_i := s_1 \cap q_2$ // see example 21
9. $I := I \cup \{s_i\}$
10. $\text{Cache}_{\{\text{post}\}, \emptyset} := \text{Cache}_{\{\text{post}\}, \emptyset} \cup \{s_i\}$
11. $s_1 := s_1 \setminus q_2$ // see example 18
12. $\leftarrow s_2, I$

To start with, the algorithm creates an empty segment $s_2$ for the remainder of query $q_2$, i.e., the part of query $q_2$ that does not overlap with any segment of the cache. Consequently, the result of the remainder query is moved into segment $s_2$. The last loaded tuple $\lambda_2$ is updated (see example 17). And the segment $s_2$ is added to the top-k semantic cache. Query $q_2$ overlaps with segment $s_1$. The algorithm calculates the intersection $s_i := s_1 \cap q_2$ of query $q_2$ and segment $s_1$ (see example 21). And segment $s_i$ is added to the top-k semantic cache as well. Finally, the algorithm subtracts query $q_2$ from the overlapping segment $s_1$ (see example 18).
Part III.

Prototype IQCache, Evaluation and Discussion
7. Implementation

The techniques described in this thesis have been implemented in a prototype of a top-k semantic cache. It is called IQCache (Intelligent Query Cache). The implementation has been documented extensively. We have carefully designed all components of the prototype using UML diagrams. We will omit them from this thesis, since all algorithms that are applied in IQCache have already been discussed in the previous chapters in great detail. Instead, we will briefly introduce the components of the prototype and showcase some interesting aspects of the implementation.

7.1. Overview

The prototype IQCache, which is implemented in Java using the integrated development environment eclipse, consists of the following components (see figure 7.1):

- **JDBC** (generic parameterizable driver)
- **Top-k semantic cache**
  - Query answering – Cache organisation – Cache replacement
- **Histograums / Pipelining**
  - Estimating query bounds
- **Satisifiability**
  - Finding overlaps / subsumptions
- **Partial materialized view management**
  - Storing and updating segments, i.e., cached top-k queries
- **Query parser and query meta data**
  - Understanding queries
- **Commons** (including Apache Commons: Lang and Math)

Figure 7.1.: Overview of components of IQCache
7. Implementation

*Commons*: This component contains classes that could be helpful in any project. It has no dependencies, except that it uses *Lang* and *Math* from the Apache Commons [Thea]. It includes utility classes for doubles, strings, dates, arrays, files and many more.

*Query parser and query meta data*: We forked the SQL parser JSqlParser [Fra], which had been implemented in Java 4. We have updated and modified this fork substantially. We have introduced generics, which have been available since Java 5. We have added new functionality, e.g., more complex *order by* clauses. This component cannot only parse, but also describe, queries. We have implemented the functionality to automatically attribute the correct meta data to parsed queries. *Insert, update, delete* and *select* queries are supported. In fact, this component even supports parameterized queries (i.e., prepared statements).

*Satisfiability*: This component contains the abstract definition of an SMT solver. In addition, the DNF solver and the TryValues solver are part of this component. The DNF solver is an SMT solver that transforms a given expression into disjunctive normal form. Then, it checks if the conjunctive parts are satisfiable using techniques for conjunctive formulas [GSW96, SB00]. The TryValues solver is another SMT solver that tries to satisfy a given expression by intelligently guessing a solution. Furthermore, this component integrates the SMT solvers Yices [DM06] (via Java Native Interface, i.e., JNI), as well as MathSAT 5 [CGSS13] and Z3 [DMBO8] (using the provided Java APIs, which equally rely on JNI) into IQCache.

*Histograms / Pipelining*: We have implemented classes for buckets, histogram creation, and histogram indexing. The utilized histograms are multidimensional equi-depth histograms. They are heuristically generated by a single SQL query using *window functions* (see appendix [A]). The latter are available in SQL since SQL:2003 [ISO03]. The algorithm getLowerBound$(H, q, \tau)$ (see algorithm 14) is also part of this component. This algorithm, which can estimate bounds for query results, allows the top-k semantic cache to pipeline query results of queries that can only be partially answered by the cache.

*Partial materialized view management*: This component organizes the management of the stored tuples of the segments of the top-k semantic cache (i.e., the $T$ of the definition of a segment, see definition 9). It uses a second JDBC connection (in contrast to the primary JDBC connection to the server). Thereby, this component supports in-memory storage, e.g., by using an in-memory database like HSQLDB [Thed], as well as disk storage, e.g., by using a fast local database like MySQL [Ora]. This component allows filtered access to the stored partial materialized views. This access supports projections, constraints, orderings and limits.

*Top-k semantic cache*: This component implements the top-k semantic cache with query processing and cache management as introduced in this thesis (see chapter 4 and chapter 6).
7.1. Overview

**JDBC**: We have designed and implemented a generic, abstract, extensible JDBC driver that is used by both the top-k semantic cache prototype IQCache and the tool JDBCWire (see below).

In addition, we have developed several tools during the design and implementation of IQCache. For example, the tool *TupleCounter* analyzes data locality in query workloads by counting the number of times each unique tuple appears in a query result. The tool *JDBCWire* can simulate JDBC connections with a specific throughput and latency. Finally, we have created an evaluation suite. Given SQL scripts for benchmarking, the suite can automatically evaluate the top-k semantic cache prototype IQCache. It maintains elaborate logs during the benchmarks. Additionally, it can automatically create plots based on the evaluation results.

Table 7.1 shows an overview of the different eclipse projects of the IQCache workspace with source lines of code (SLOC) and comment lines of code (CLOC).

<table>
<thead>
<tr>
<th>component</th>
<th>eclipse project</th>
<th>SLOC (CLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>JDBC</td>
<td>2485 (2271)</td>
</tr>
<tr>
<td>top-k semantic cache</td>
<td>Semantic Cache</td>
<td>6424 (5757)</td>
</tr>
<tr>
<td>histograms / pipelining</td>
<td>Histogram</td>
<td>5759 (4830)</td>
</tr>
<tr>
<td>satisfiability</td>
<td>Satisfiability,</td>
<td>2720 (2546)</td>
</tr>
<tr>
<td></td>
<td>SMTSolver,</td>
<td>7675 (2922)</td>
</tr>
<tr>
<td></td>
<td>Yices</td>
<td>2834 (1755)</td>
</tr>
<tr>
<td>materialized view management</td>
<td>Physical Cache</td>
<td>2291 (2462)</td>
</tr>
<tr>
<td>query parser and</td>
<td>JSQLParse,</td>
<td>10286 (4001)</td>
</tr>
<tr>
<td>query meta data</td>
<td>Query</td>
<td>18930 (19835)</td>
</tr>
<tr>
<td>commons</td>
<td>Commons</td>
<td>16059 (18532)</td>
</tr>
<tr>
<td>evaluation</td>
<td>Evaluation,</td>
<td>9080 (6801)</td>
</tr>
<tr>
<td></td>
<td>LittleE,</td>
<td>1619 (1673)</td>
</tr>
<tr>
<td></td>
<td>Test</td>
<td>116 (228)</td>
</tr>
<tr>
<td>tool TupleCounter</td>
<td>TupleCounter</td>
<td>231 (92)</td>
</tr>
<tr>
<td>tool JDBCWire</td>
<td>JDBCWire</td>
<td>3641 (3638)</td>
</tr>
<tr>
<td></td>
<td>JDBCWire DelayTests</td>
<td>2007 (2730)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5879 (6460)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>92159 (81891)</td>
</tr>
</tbody>
</table>

Table 7.1: Different eclipse projects of the IQCache workspace with source lines of code (SLOC) and comment lines of code (CLOC)
7. Implementation

7.2. Best Cache Replacement Strategy

In the evaluation, we have used the LRU replacement policy in the top-k semantic cache, because it is the simplest one and a good baseline (see section 8.3). In his 2012 bachelor’s thesis “Evaluation of Replacement Algorithms for Usage in a Semantic Cache” \cite{Buer12}, Florian Büchner implemented the cache replacement policies MRU, PRAG \cite{LKRPM01}, EXP1 \cite{RF98}, GD* \cite{JB01}, INTER/INTRA \cite{JB01}, and TSP \cite{YZ01}. He evaluated the different replacement strategies using the top-k semantic cache prototype IQCache. He has discovered that the top-k semantic cache performs extremely well using the LRU replacement policy. The experiments show that the best replacement strategies are PRAG and GD*, which both significantly outperform LRU. Overall, GD* yielded the best results and managed to beat PRAG for first place.

7.3. Best Histograms for the Estimation of Query Bounds

As discussed in section 5.1 multidimensional histograms can consist of a combination of a multidimensional structure and a one-dimensional partition algorithm. In his 2014 bachelor’s thesis “Evaluation of Multidimensional Histograms for the Estimation of Bounds of Ordered Queries” \cite{Rau14}, Sven Rausch evaluated the suitability for the estimation of query bounds of the different combinations of multidimensional structures and one-dimensional partition algorithms using the top-k semantic cache prototype IQCache. He concluded that multidimensional equi-depth histograms are suited best. As a matter of fact, IQCache uses multidimensional equi-depth histograms. For future work, we plan to research histograms that analyze the workload. By covering hot regions with a denser structure, i.e., more buckets, we expect the histogram to achieve, on average, a better estimation of the bounds of query results.

In our experiments, we have studied the utilization of targeted histograms that only cover a specified hot region of the underlying base table (see section 8.4.2.6 and section 8.5.2.6).
8. Evaluation

In this chapter, we extensively and thoroughly evaluate top-k semantic caching using the prototype IQCache and the benchmarks Yahoo! Cloud Serving Benchmark, the Twitter benchmark, and some test queries from the MonArch 3 system. But first, we explain the selection of these three benchmarks, the different utilized caches and the set-up of the evaluation.

8.1. Benchmark Selection

We want to select several benchmarks to evaluate top-k semantic caching using the prototype IQCache. But, to be able to actually get meaningful results, we need the selected benchmarks to fulfill the following requirements:

1. **Supported queries**: The majority of the select queries that are issued by the benchmark must be supported by the top-k semantic cache (or rather, its prototype IQCache). Basically, this means that the used select queries must be select-project-join queries. They may be ordered or limited. But we will also consider benchmarks that use queries that are not sorted, not limited, or neither sorted nor limited. If the queries of the benchmark are not sorted, we will add a fixed ordering.

2. **Sizable query result**: The results of the select queries of the benchmark must contain a sizable amount of tuples. Top-k semantic caching only makes sense if overlaps of queries can be exploited. This will only be possible if the query results usually contain more than one tuple.

3. **Data locality**: There must be locality in the results of the select queries of the benchmark, i.e., there is a certain percentage of tuples that is contained in the query results of many queries.

We have written the tool *TupleCounter* to analyze data locality of benchmarks. It counts, for each tuple, how often that tuple appears in a query result during execution of the benchmark. We call this number the *tuple count* of the tuple. By enumerating all tuple counts in descending order, we obtain a graph that accurately describes the data locality of the examined benchmark.

We considered the following benchmarks to evaluate the top-k semantic cache:
### 8. Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuctionMark</td>
<td>The AuctionMark benchmark is an OLTP benchmark. It simulates an online auction and shopping website [AP, APSZ12].</td>
</tr>
<tr>
<td>epinions</td>
<td>The epinions benchmark is based on data and statistics from the consumer review website epinions.com [MA05].</td>
</tr>
<tr>
<td>LinkBench</td>
<td>The LinkBench benchmark simulates workloads similar to those of Facebook's production deployment. Therefore, it simulates queries on a social graph [APBC13].</td>
</tr>
<tr>
<td>MonArch 3</td>
<td>A test based on queries that were recorded during two typical user sessions on the MonArch 3 prototype [SWF11].</td>
</tr>
<tr>
<td>ResourceStresser</td>
<td>The ResourceStresser benchmark is a synthetic benchmark that can do isolated stress testing on CPU, disk I/O and locks of a database system [CDPCM12].</td>
</tr>
<tr>
<td>SeatS</td>
<td>The SeatS benchmark simulates “an airline ticketing system, where customers search for flights and make online reservations” [SP].</td>
</tr>
<tr>
<td>TATP</td>
<td>The Telecommunication Application Transaction Processing (TATP) benchmark simulates a typical Home Location Register (HLR) database used by a mobile carrier. [Wol09]</td>
</tr>
<tr>
<td>TPC-C</td>
<td>The TPC-C benchmark, an OLTP benchmark, simulates an order processing system. The benchmark covers “entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouse” [LD93, Raa10, RKS].</td>
</tr>
<tr>
<td>TPC-E</td>
<td>The TPC-E benchmark, another OLTP benchmark, simulates the workload of “a brokerage firm with customers who generate transactions related to trades, account inquiries, and market research” [CAA+11].</td>
</tr>
<tr>
<td>Twitter</td>
<td>The Twitter benchmark simulates a micro-blogging platform. It is designed using “an anonymized snapshot of the Twitter social graph from August 2009 that contains 51 million users and almost 2 billion ‘follows’ relationships” [CHBG10].</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>The Wikipedia benchmark simulates the online encyclopedia Wikipedia. It uses “the real schema, transactions, and queries as the live website” [UPvS09, CDPCM12].</td>
</tr>
<tr>
<td>YCSB</td>
<td>The Yahoo! Cloud Serving Benchmark is actually a collection of benchmarks. It contains “various combinations of read/write operations and access distributions that match products inside Yahoo!” [CST+10].</td>
</tr>
</tbody>
</table>
After analysis of the benchmarks, we have discarded the benchmarks AuctionMark, epinions, LinkBench, SeatS, TATP, TPC-C, TPC-E, and Wikipedia, because the sizes of query results of these benchmarks are too small. For example, on average, the TPC-C benchmark issues 11 percent queries with empty results, 86 percent queries that return exactly one tuple and only about 3 percent queries that return more than one tuple. As another example, on average, the AuctionMark benchmark creates 15 percent queries with empty results, 83.5 percent queries that return exactly one tuple and merely 1.5 percent queries that return more than a single tuple. But if only 3 percent or 1.5 percent, respectively, of the queries of the benchmark are interesting for top-k semantic caching, the benchmark itself should not be used to evaluate a top-k semantic cache.

In addition, the synthetic benchmark ResourceStresser misses the needed data locality and will therefore not be used.

Fortunately, three benchmarks do fulfill our requirements: the Yahoo! Cloud Serving Benchmark using a ScanRecord workload \([\text{CST}^{+10}]\), the Twitter benchmark using a GetTweetsFromFollowing workload \([\text{CHBG10}]\), and the MonArch 3 test queries \([\text{SWF11}]\).

In this chapter, we use these three benchmarks to thoroughly evaluate top-k semantic caching using the prototype IQCache.
8. Evaluation

8.2. Caches

In the evaluation, the following seven different caches will be utilized:

1. The hash cache \((HC)\) simply hashes the query string and stores the corresponding query result in a hash map. Therefore, it can return a cached query result if the hash map contains an exact match to the request. This cache uses the LRU cache replacement policy. The hash cache is the baseline that the logical and semantic caches must beat.

2. The top-k semantic cache with pipelining \((SC+t)\) uses all techniques of top-k semantic caching as described in this thesis.

3. The top-k semantic cache \((SC-)\) uses the techniques of top-k semantic caching as described in this thesis. But it does not estimate a lower bound for the query result of the remainder query. Therefore, if a query can only partially be answered by the top-k semantic cache, it will have to wait for the first tuple from the server to answer the query.

4. The any-k semantic cache \((SC+a)\) uses the techniques of top-k semantic caching as described in this thesis. But if it answers a query, it will ignore the \texttt{order by}. Hence, there is no need to estimate a lower bound for the query result of the remainder query. The any-k semantic cache can just return all locally available tuples. This cache can act as a base line of what the top-k semantic cache with pipelining \((SC+t)\) can maximally achieve in comparison with the top-k semantic cache that does not estimate a lower bound for the query result of the remainder query \((SC-)\).

5. The top-k logical cache with pipelining \((LC+t)\) uses most techniques of top-k semantic caching, but discards the disjointness property of the top-k semantic cache. It allows that segments overlap. Hence, the same tuples may be contained in multiple segments.

6. The top-k logical cache \((LC-)\) is a top-k logical cache that does not estimate a lower bound for the query result of the remainder query.

7. The any-k logical cache \((LC+a)\) is the any-k variant of the top-k logical cache.

The prototype IQCache is able to act as a top-k semantic cache with and without pipelining, as an any-k semantic cache, as well as a top-k logical cache with and without pipelining, and as a any-k logical cache.
8.3. Set-up

We have used the following evaluation set-up:

Platform: We have run our experiments on an Intel Core i7 CPU with 8GB RAM. The operation system has been Windows 7 Professional. We have used the Java SE Development Kit 6u45 to run the prototype and evaluation environment.

Connection: We have simulated connections with different server latencies and throughputs (see table 8.2). While holding the server latency at 50ms, we have simulated bandwidths from 128kbit/s up to 10Mbit/s. On the other hand, with a fixed throughput of 1Mbit/s, we have simulated server latencies from 1ms up to 100ms, or round trip times up to 200ms, respectively.

We have written the tool JDBCWire that is able to simulate additional propagation delays, i.e., server latency, and reduced bandwidth, i.e., throughput, of JDBC connections for measurement purposes. JDBCWire itself is a JDBC driver that can be inserted between an application and a JDBC driver. It supports constant delays and probability distributions (e.g., normal, exponential and uniform distributions). It achieves microsecond precision. In our experiments, we have only used constant delays.

<table>
<thead>
<tr>
<th>domain</th>
<th>default</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>server latency (in ms)</td>
<td>50</td>
<td>1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>128, 256, 512, 768, 1024, 1536, 2048, 2560, 3072, 3584, 4096, 4608, 5120,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6144, 7168, 8192, 9216, 10240</td>
</tr>
<tr>
<td>throughput (in kbit/s)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2.: Utilized values of the different domains

Processing: To be able to observe the effects of pipelining, we have assumed a processing time of 5ms per tuple by the client application.

Database instance: We have used the Yahoo! Cloud Serving Benchmark with a ScanRecord workload [CST+10], the Twitter benchmark with a GetTweetsFromFollowing workload [CHBG10], and the MonArch 3 test queries [SWF11].

For the Yahoo! Cloud Serving Benchmark and the Twitter benchmark, we have used database instances with 100 000 to 1 000 000 tuples. The MonArch 3 test queries run on an actual instance from the MonArch project with various tables. The relevant tables for the Monarch 3 test queries contain 11 017 tuples (all tables contain 33 519). Table 8.3 shows an overview.

Cache constraints: For each test run, we restrict the number of tuples in the cache (see table 8.4). At the beginning of a test run, the cache is always empty. Therefore, it needs
8. Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>scales (i.e., number of tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>YCSB</td>
<td>100 000, 200 000, \ldots, 1 000 000</td>
</tr>
<tr>
<td>Twitter</td>
<td>100 000, 200 000, \ldots, 1 000 000</td>
</tr>
<tr>
<td>MonArch 3</td>
<td>11 017 (33 519 total) for both of the tests</td>
</tr>
</tbody>
</table>

Table 8.3.: Used scales of the benchmarks

some ramp-up time to achieve its full potential. The actual cache sizes are dependent on the scale, i.e., the number of tuples in the database instance.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>cache sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>YCSB</td>
<td>0.2, 0.4, \ldots, 1.0, 2.0, \ldots, 5.0 percent</td>
</tr>
<tr>
<td>Twitter</td>
<td>0.05, 0.1, \ldots, 0.5 percent</td>
</tr>
<tr>
<td>MonArch 3</td>
<td>100, 200, \ldots, 1500 or 2500 tuples</td>
</tr>
</tbody>
</table>

Table 8.4.: Used cache sizes of the benchmarks

Workload size: A load of the Yahoo! Cloud Serving Benchmark consists of about 10 000 queries. We used a load of the Twitter benchmark that issues about 2 500 queries. There are two sets of MonArch 3 test queries. The first set consists of 199 queries and the second one of 212 queries.

Reproducibility: For each configuration, all benchmarks have been executed at least three times. The Yahoo! Cloud Serving Benchmark has even been run ten times for each configuration.

Replacement strategy: In this evaluation, the semantic cache always uses a modified LRU replacement policy. We have chosen LRU, because it is the simplest one and a good baseline. But we have needed two alterations to the plain LRU replacement policy to enhance cache performance.

First, cache segments that are very small, e.g., less than 5 tuples, are removed. This reduces the number of cache segments in the cache. Accordingly, this policy reduces the number of considered and used segments during query processing. Hence, the first alteration reduces the number of calls to the SMT solver.

Secondly, cache segments with a description that is too complex, because many other segments have been subtracted from it, e.g., more than 5, are removed from the cache as well. This second alteration ensures that the expressions that the SMT solver must solve do not take very long to solve.

Satisfiability checks: In the semantic and logical caches, an SMT solver – actually the hybrid SMT solver that has been discussed in section 3.2.2 – is used to check the satisfi-
ability of expressions, which arise during query processing and cache reorganization. This takes time and has an impact on cache performance. Therefore, we have analyzed the effect of the average duration of a single satisfiability check. Table 8.5 contains a list of all considered durations.

<table>
<thead>
<tr>
<th>domain</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>average duration of a single satisfiability check (in ms)</td>
<td>0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 3.75, 4, 4.25, 4.5, 4.75, 5</td>
</tr>
<tr>
<td>estimation performance (as ratio)</td>
<td>0, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.175, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1</td>
</tr>
</tbody>
</table>

Table 8.5.: Utilized values of the analyzed domains

**Histogram selection**: Optimal histogram selection is NP-complete (see corollary 20). Hence, in our prototype IQCache, currently, the user has to define the utilized histograms. In our experiments, we have created histograms to cover all queries. We have used histograms of appropriate size with relation to scale (e.g., number of buckets is about 1 percent of scale in the Yahoo! Cloud Serving Benchmark) to achieve a constantly good estimation performance for all scales (see below). In section 8.4.2.6 and section 8.5.2.6, we further discuss the required size of the selected histograms.

**Estimation performance**: The quality of the estimation of the lower bound of the query result of the remainder query may play an important role in the effectiveness of pipelining queries that can only be partially answered by the cache. For each query, we can describe the quality as the ratio of the estimated position to the actual position of the first tuple from the server. We call this ratio the estimation performance. After recording the actual positions of the first tuples from the servers in the queries of the benchmark, we have analyzed the effect of different estimation performances on the relative query execution time. The relative query execution time is the cumulative execution time of all queries of the benchmark using the cache divided by the cumulative execution time of all queries of the benchmark without cache usage. We have studied the full range from 0 to 1, which an estimation performance can have (see table 8.5).

**Analyses**: Table 8.6 shows an overview of the analyses of the evaluation.
8. Evaluation

<table>
<thead>
<tr>
<th>#</th>
<th>scope</th>
<th>domain</th>
<th>co-domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>for each scale</td>
<td>cache size</td>
<td>hit rate</td>
</tr>
<tr>
<td>2.</td>
<td>for each scale</td>
<td>cache size</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>3.</td>
<td>for each cache size of each scale</td>
<td>server latency</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>4.</td>
<td>for each cache size of each scale</td>
<td>throughput</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>5.</td>
<td>for each cache size of each scale</td>
<td>average duration of a single satisfiability check</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>6.</td>
<td>for each cache size of each scale</td>
<td>estimation performance</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>7.</td>
<td>for each ratio of cache size and scale</td>
<td>scale</td>
<td>hit rate</td>
</tr>
<tr>
<td>8.</td>
<td>for each ratio of cache size and scale</td>
<td>scale</td>
<td>relative query execution time</td>
</tr>
<tr>
<td>9.</td>
<td>for each ratio of cache size and scale</td>
<td>scale</td>
<td>number of considered segments</td>
</tr>
<tr>
<td>10.</td>
<td>for each ratio of cache size and scale</td>
<td>scale</td>
<td>number of used segments</td>
</tr>
</tbody>
</table>

Table 8.6.: Overview of analyses of the evaluation
8.4. Twitter Benchmark

The Twitter benchmark is based on the popular micro-blogging platform twitter.com. The authors of this benchmark were able to obtain an "an anonymized snapshot of the Twitter social graph from August 2009 that contains 51 million users and almost 2 billion ‘follows’ relationships" [CHBG10]. Thereby, they were able to create a realistic benchmark.

8.4.1. Description

The Twitter benchmark is configurable. It supports a mixture of the following workloads: GetFollowers returns all followers of a user. GetTweet returns a specific tweet. GetTweetsFromFollowing returns all tweets of other users that a user is following. GetUserTweets returns all tweets of a user. InsertTweet inserts a new tweet for a user.

We have used the Twitter benchmark with a GetTweetsFromFollowing workload, because it is particularly interesting for caching: If two users are following the same other user, both of them will need the tweets of this other user. Hence, their corresponding query results will very likely overlap. Listing 8.1 shows a typical query of this workload. In this example, \{0, 1, 2, 4, 5, 15, 84\} is the set of ids of users that the current user is following.

```
1 SELECT *
2 FROM tweets t
3 WHERE t.uid IN (0, 1, 2, 4, 5, 15, 84)
4 ORDER BY t.uid, t.id
5 LIMIT 100
```

Listing 8.1: Query \(q_{twitter}\): a typical query of the Twitter benchmark

One run of the Twitter benchmark consisted of approximately 2 500 queries that were similar to this example query. We have evaluated 10 different scales. In the smallest, the main table tweets contained 100 000 tuples. In the largest, it was populated by 1 000 000 tuples.

8.4.2. Evaluation

The results of the evaluation are very similar for the different scales. As an example, in the presentation and discussion of the evaluation, we will focus on the scale of 400 000 tuples.

8.4.2.1. Cache Size and Hit Rate

To start with, we look at cache size versus hit rate for the scale of 400 000 tuples (see figure 8.1).

*Expected behavior*: If the cache size is increased, the hit rate will rise.
8. Evaluation

![Figure 8.1: Twitter: Cache size versus hit rate for the scale of 400,000 tuples](image)

**Observed behavior and discussion**: The hit rate rises for small cache sizes, but soon reaches plateau at a hit rate of almost 80 percent and a cache size of 800 tuples. Apparently, a cache size of more than 800 tuples cannot increase the hit rate much further. The hash cache can only achieve less than half the hit rate of the semantic caches and logical caches.

### 8.4.2.2. Cache Size and Execution Time

Next, we study the cache size versus relative query execution time for the scale of 400,000 tuples (see figure 8.2).

**Expected behavior**: If the cache size is increased, the relative query execution time will fall until a plateau is reached.

**Observed behavior and discussion**: For smaller cache sizes, an increased cache size does lead to a smaller relative query execution time. But with larger cache sizes, the relative query execution time starts to slowly rise again, because the cache contains more segments that have to be considered during query processing. There is an optimal cache size at about 800 tuples. Therefore, we will use a cache size of 800 from now on.

### 8.4.2.3. Server Latency and Execution Time

For the cache size of 800 tuples and the scale of 400,000 tuples, we investigate the server latency versus relative query execution time (see figure 8.3).

**Expected behavior**: Caching is more effective for higher server latencies.
8.4. Twitter Benchmark

Figure 8.2.: Twitter: Cache size versus relative query execution time for the scale of 400,000 tuples

Figure 8.3.: Twitter: Server latency versus relative query execution time for cache size of 800 tuples and the scale of 400,000 tuples
8. Evaluation

*Observed behavior and discussion:* Caching is indeed more effective for higher latencies, but only by a little. Apparently, caching is effective for small and high latencies.

### 8.4.2.4. Throughput and Execution Time

Next, we examine throughput versus relative query execution time for the cache size of 800 tuples and the scale of 400,000 tuples (see figure 8.4).

![Figure 8.4: Twitter: Throughput versus relative query execution time for cache size of 800 tuples and the scale of 400,000 tuples](image)

*Expected behavior:* Caching is more effective for lower throughputs.

*Observed behavior and discussion:* Caching is really effective for lower throughputs, especially for throughputs lower than 1024 kbps.

### 8.4.2.5. Satisfiability Checks and Execution Time

We have analyzed the average duration of a single satisfiability check versus relative query execution time for the cache size of 800 tuples and the scale of 400,000 tuples (see figure 8.5).

*Expected behavior:* If the average duration of a single satisfiability check is higher, the relative query execution time is also higher.

*Observed behavior and discussion:* The relationship between the average duration of a single satisfiability check and the relative query execution time is linear. To achieve a significant effect on the relative query execution time using top-k semantic cache or top-k logical cache, an average duration of a single satisfiability check of less than 1 ms...
8.4. Twitter Benchmark

Figure 8.5.: Twitter: Average duration of a single satisfiability check versus relative query execution time for cache size of 800 tuples and the scale of 400,000 tuples would be desirable. The hybrid solver achieves an average duration of less than 0.5ms (see section 3.2.2).

8.4.2.6. Estimation Performance and Execution Time

We have also analyzed the estimation performance versus relative query execution time for the cache size of 800 tuples and the scale of 400,000 tuples (see figure 8.6). We will consider only pipelined queries, i.e., queries that could only be partially answered by the top-k cache. The top-k semantic cache without pipelining (SC-) and the any-k semantic cache (SC+a) serve as worst case or best case baselines, respectively.

Expected behavior: If the estimation performance is better, the relative query execution time will also be better.

Observed behavior and discussion: For an estimation performance of 0, the top-k semantic cache with (SC+t) and without (SC-) pipelining achieve the same relative query execution time. That is to be expected, as the top-k semantic cache calculates the bound for the result of the remainder query while it waits for the server response. Therefore, this calculation does not negatively impact cache performance.

As the estimation performance increases, the relative query execution time falls. An estimation performance of only about 50 percent of the top-k semantic cache is sufficient to achieve a significant improvement in comparison to the top-k semantic cache without pipelining.

At perfect estimation of the lower bound of the result of the remainder query, the
Figure 8.6.: Twitter: Estimation performance versus relative query execution time for cache size of 800 tuples and the scale of 400,000 tuples (considering only pipelined queries).

The performance of the top-k semantic cache almost reaches the performance of the any-k semantic cache (SC+a) that completely ignores orderings and can thus return all locally cached tuples.

To achieve an estimation performance of 50 percent, we need a histogram that uses about 1,800 buckets to describe the relation tweets (see figure 8.7). This is a huge overhead in relation to cache size as we have established that a cache size of 800 tuples is sufficient.

We can reduce the number of buckets that are needed by identifying the hot region of the relation tweets. The hot region of a relation is a sub-set of tuples of the relation that is accessed significantly more by a given workload.

We have analyzed which uids are referenced more than others in the queries of the workload. Using those hot uids, we have been able to describe a hot region that contains about 8,000 tuples (i.e., 2 percent of scale). We have utilized a histogram that only targets this hot region. Of course, using this approach, the targeted histogram can only be used to estimate bounds of results of queries that are contained within the hot region (i.e., the expression that describes the targeted hot region of the histogram must subsume the condition of the query). Nevertheless, the targeted histogram has been able to achieve an estimation performance of over 50 percent for just 35 buckets (see figure 8.8). If we are willing to invest into 50 buckets, we can even get an estimation performance of almost 80 percent.

In conclusion, using a targeted histogram, we can drastically reduce the overhead and,
8.4. Twitter Benchmark

Figure 8.7.: Twitter: Number of buckets versus estimation performance for cache size of 800 tuples and the scale of 400,000 tuples (full histogram)

Figure 8.8.: Twitter: Number of buckets versus estimation performance for cache size of 800 tuples and the scale of 400,000 tuples (targeted histogram)
8. Evaluation

furthermore, significantly improve the estimation performance.

8.4.2.7. Scale and Hit Rate

After only discussing a specific scale so far, we will now look at the cache behavior over all evaluated scales. To begin with, we study scale versus hit rate for the cache size of 0.2 percent of scale (see figure 8.9).

![Figure 8.9: Twitter: Scale versus hit rate for the cache size of 0.2 percent of scale](image)

*Expected behavior*: The hit rate is the same for all scales.
*Observed behavior and discussion*: As expected, the hit rate is mostly the same for all scales.

8.4.2.8. Scale and Execution Time

Next, we consider scale versus relative query execution time for the cache size of 0.2 percent of scale (see figure 8.10 and figure 8.11).

*Expected behavior*: Caching is more effective for smaller scales.
*Observed behavior and discussion*: Actually, caching is equally effective for small and large scales.

8.4.2.9. Scale and Number of Considered Segments

We investigate scale versus number of considered segments for the cache size of 0.2 percent of scale (see figure 8.12).
Figure 8.10.: Twitter: Scale versus relative query execution time for the cache size of 0.2 percent of scale (top-k cache types)

Figure 8.11.: Twitter: Scale versus relative query execution time for the cache size of 0.2 percent of scale (all cache types)
8. Evaluation

Figure 8.12.: Twitter: Scale versus number of considered segments for the cache size of 0.2 percent of scale

*Expected behavior:* The number of considered segments rises linearly in relation to scale (or cache size, respectively).

*Observed behavior and discussion:* There is indeed a linear relationship between the number of considered segments and scale (or cache size). In addition, we can observe that, for large scales, the top-k semantic cache considers more segments than the top-k logical cache.

### 8.4.2.10. Scale and Number of Used Segments

Finally, we examine scale versus number of used segments for the cache size of 0.2 percent of scale (see figure 8.13).

*Expected behavior:* The number of used segments rises linearly in relation to scale (or cache size, respectively).

*Observed behavior and discussion:* The number of used segments of the top-k logical cache does increase linearly with the scale. But the number of used segments of the top-k semantic cache does not. On average, it just needs 2 segments (standard deviation: 2.3) to answer a query.
Figure 8.13.: Twitter: Scale versus number of used segments for the cache size of 0.2 percent of scale
8. Evaluation

8.5. Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark is a collection of benchmarks that contain “various combinations of read/write operations and access distributions that match products inside Yahoo!” \cite{CST10}. The benchmark can simulate primary-key based key-value store applications with simple workload that requires high scalability. Hence, it has been originally designed to evaluate key-value stores, but can also be useful to study traditional database systems.

8.5.1. Description

The Yahoo! Cloud Serving Benchmark always operates on only one table, which it calls \texttt{USERTABLE}. This table contains all tuples. But the benchmark does support various workloads: \texttt{DeleteRecord} deletes a record. \texttt{InsertRecord} inserts a tuple. \texttt{Read-ModifyWriteRecord} selects a tuple for modification, modifies the record, and writes it back. \texttt{ReadRecord} reads a tuple. \texttt{ScanRecord} reads a range of tuples. \texttt{UpdateRecord} updates a tuple.

We have used the Yahoo! Cloud Serving Benchmark with a \texttt{ScanRecord} workload, because only this workload consists of queries with results that contain more than one tuple. Therefore, only this workload can potentially possess overlaps and subsumptions of query results. Listing 8.2 shows a typical query of this workload.

```
1 SELECT *
2 FROM "USERTABLE" u
3 WHERE u.ycsb_key > 228863 AND u.ycsb_key < 229080
4 ORDER BY u.ycsb_key
```

Listing 8.2: Query \texttt{qycsb}: a typical query of the Yahoo! Cloud Serving Benchmark

During one run, the Yahoo! Cloud Serving Benchmark executed about 10,000 queries. We have considered 10 different scales. The relation \texttt{USERTABLE} contained 100,000 tuples in its smallest incarnation. The largest one contained 1,000,000 tuples.

8.5.2. Evaluation

We will exemplify the evaluation on the scale of 800,000 tuples. Generally, results of the evaluation are very similar for the different scales.

8.5.2.1. Cache Size and Hit Rate

To begin with, we take a look at cache size versus hit rate for the scale of 800,000 tuples (see figure 8.14).

\textit{Expected behavior:} If the cache size is increased, the hit rate will rise.
8.5. Yahoo! Cloud Serving Benchmark

![Diagram](image_url)

**Figure 8.14.:** YCSB: Cache size versus hit rate for the scale of 800,000 tuples

*Observed behavior and discussion:* As expected, the hit rate of both the semantic caches and the logical caches rises with increasing cache size. But the slope is much steeper than anticipated. Even for small cache sizes, a very good hit rate can be achieved. On the other hand, because the bounds of the range queries of the Yahoo! Cloud Serving Benchmark vary too much, the hash cache is not suited for the workload of this benchmark.

### 8.5.2.2. Cache Size and Execution Time

Next, we study the cache size versus relative query execution time for the scale of 800,000 tuples (see figure 8.14).

*Expected behavior:* If the cache size is increased, the relative query execution time will fall until a plateau is reached.

*Observed behavior and discussion:* At the beginning, an increased cache size does indeed yield a smaller relative query execution time. But, especially for the top-k logical cache, with larger cache sizes, the relative query execution time starts to rise again, because the cache contains more segments that are possibly relevant for query processing. We will use a cache size of 8,000 henceforth.

### 8.5.2.3. Server Latency and Execution Time

For the cache size of 8,000 tuples and the scale of 800,000 tuples, we investigate the server latency versus relative query execution time (see figure 8.16).
8. Evaluation

Figure 8.15.: YCSB: Cache size versus relative query execution time for the scale of 800 000 tuples

Figure 8.16.: YCSB: Server latency versus relative query execution time for the cache size of 8000 tuples and the scale of 800 000 tuples
8.5. Yahoo! Cloud Serving Benchmark

Expected behavior: Caching is more effective for higher server latencies. Observed behavior and discussion: Actually, caching is equally effective for small and high latencies. The throughput seems to be the limiting factor of a bad connection.

8.5.2.4. Throughput and Execution Time

Therefore, we examine throughput versus relative query execution time for the cache size of 8 000 tuples and the scale of 800 000 tuples next (see figure 8.17).

![Graph](image)

Figure 8.17.: YCSB: Throughput versus relative query execution time for the cache size of 8 000 tuples and the scale of 800 000 tuples

In addition, we will also consider throughput versus relative query execution time for the cache size of 40 000 tuples (see figure 8.18). Expected behavior: Caching is more effective for lower throughputs. Observed behavior and discussion: Caching is indeed more effective for lower throughputs. It is more pronounced for very large cache sizes. For extremely low throughputs, the evaluation shows that larger cache sizes are preferable. They are able to achieve a better relative query execution time.

8.5.2.5. Satisfiability Checks and Execution Time

We have analyzed the average duration of a single satisfiability check versus relative query execution time for the cache size of 8 000 tuples and the scale of 800 000 tuples (see figure 8.19).
8. Evaluation

![Graph](image)

Figure 8.18.: YCSB: Throughput versus relative query execution time for the cache size of 40,000 tuples and the scale of 800,000 tuples

![Graph](image)

Figure 8.19.: YCSB: Average duration of a single satisfiability check versus relative query execution time for the cache size of 8,000 tuples and the scale of 800,000 tuples
**8.5. Yahoo! Cloud Serving Benchmark**

*Expected behavior:* If the average duration of a single satisfiability check is higher, the relative query execution time is also higher.

*Observed behavior and discussion:* There is a linear relationship between the average duration of a single satisfiability check and the relative query execution time. The average duration of a single satisfiability check should be less than about 5ms to allow for effective caching using a top-k semantic cache or top-k logical cache. Fortunately, the hybrid solver is able to solve them in less than 0.1ms (see section 3.2.2).

### 8.5.2.6. Estimation Performance and Execution Time

We have also analyzed the estimation performance versus relative query execution time for the cache size of 8 000 tuples and the scale of 800 000 tuples (see figure 8.20). We will consider only pipelined queries, i.e., queries that could only be partially answered by the top-k cache. The top-k semantic caches with (SC+t) and without (SC-) pipelining serve as worst case or best case baselines, respectively.

![Figure 8.20](image.png)

*Figure 8.20:* YCSB: Estimation performance versus relative query execution time for the cache size of 8 000 tuples and the scale of 800 000 tuples (considering only pipelined queries)

*Expected behavior:* If the estimation performance is better, the relative query execution time will also be better.

*Observed behavior and discussion:* The top-k semantic caches with (SC+t) and without (SC-) pipelining have the same relative query execution time for an estimation performance of 0. Because the calculation of the bound of the result of the remainder query takes place during the time that the
8. Evaluation

cache waits for the answer of the server, this calculation cannot hinder the performance of the cache. Hence, this behavior is the expected one.
The relative query execution time falls rapidly with increasing estimation performance. At an estimation performance of 30 percent, the top-k semantic caches almost matches the any-k cache that completely ignores orderings and can thus return all locally cached tuples in performance.
Generally, the effect of pipelining is smaller than in the previously discussed Twitter benchmark, but it is still noticeable.
We have analyzed how many buckets are needed to achieve a good estimation performance (see figure 8.21). The results show that, for example, a histogram of 8 000 buckets (i.e., 1 percent of scale) achieves an estimation performance of more than 50 percent. This is well above the needed estimation performance of 30 percent.

![Figure 8.21: YCSB: Number of buckets versus estimation performance for cache size of 8 000 tuples and the scale of 800 000 tuples (full histogram)](image)

We have also studied possible number of buckets for different scales. The number of buckets needed to get a certain estimation performance has proven to be very consistent. While a number of buckets equal to 0.5 percent of scale is enough to maintain an estimation performance of over 30 percent, a number of buckets equal to 1 percent is able to get a very good estimation performance for all scales of about 50 percent (see figure 8.22).

4 000 to 8 000 buckets represent a huge overhead for a cache size of just 8 000 tuples. Hence, we have identified a hot region of the relation USERTABLE, i.e., a sub-set of tuples of the relation that is accessed significantly more by the given workload. This hot region consists of 16 000 tuples (i.e., 1 percent of scale). We have described the
hot region using a logical expression. And we have constructed a histogram that only targets the hot region. But keep in mind that the targeted histogram can only be used to estimate bounds of results of queries that are contained within the hot region. We can check if the histogram covers a given query by using the SMT solver to work out if the expression that describes the targeted hot region of the histogram subsumes the condition of the query.

Even though the targeted histogram has an estimation performance of 0 percent for queries that are not covered by it, the targeted histogram achieves an estimation performance of almost 50 percent for just 100 buckets and of more than 70 percent for 400 buckets (see figure 8.23).

Similar to the untargeted histogram, the targeted histogram is very consistent with regard to scale. A targeted histogram with a number of buckets equal to just 0.05 percent of scale achieves estimation performances around 70 percent for all scales (see figure 8.24).

Based on these results, we can conclude that we can dramatically reduce the overhead and significantly improve the estimation performance using targeted histograms for the estimation of bounds of results of queries.

8.5.2.7. Scale and Hit Rate

Until now, we have only discussed a specific scale. Henceforth, we will look at the cache behavior over all evaluated scales. To begin with, we study scale versus hit rate for the
8. Evaluation

Figure 8.23.: YCSB: Number of buckets versus estimation performance for cache size of 8,000 tuples and the scale of 800,000 tuples (targeted histogram)

Figure 8.24.: YCSB: Scale versus estimation performance for cache size of 1 percent of scale and for number of buckets of 0.05 percent of scale (targeted histogram)
8.5. Yahoo! Cloud Serving Benchmark

cache size of 1 percent of scale (see figure 8.25).

![Graph of hit rate vs scale]

Figure 8.25.: YCSB: Scale versus hit rate for the cache size of 1 percent of scale

**Expected behavior:** The hit rate is the same for all scales.

**Observed behavior and discussion:** The hit rates for all scales of more than 300,000 tuples are the same. But small scales of the benchmark apparently need higher relative cache sizes to achieve the same hit rate as the larger ones.

8.5.2.8. Scale and Execution Time

Next, we study scale versus relative query execution time for the cache size of 1 percent of scale (see figure 8.26 and figure 8.27).

**Expected behavior:** Caching is more effective for smaller scales.

**Observed behavior and discussion:** As a matter of fact, caching is equally effective for small and large scales.

8.5.2.9. Scale and Number of Considered Segments

We examine scale versus number of considered segments for the cache size of 1 percent of scale (see figure 8.28).

**Expected behavior:** The number of considered segments rises linearly in relation to scale (or cache size, respectively).

**Observed behavior and discussion:** The number of considered segments does rise linearly in relation to scale (or cache size). For large scales, the top-k semantic cache
8. Evaluation

Figure 8.26: YCSB: Scale versus relative query execution time for the cache size of 1 percent of scale (top-k cache types)

Figure 8.27: YCSB: Scale versus relative query execution time for the cache size of 1 percent of scale (all cache types)
8.5. Yahoo! Cloud Serving Benchmark

Figure 8.28.: YCSB: Scale versus number of considered segments for the cache size of 1 percent of scale

considers less segments than the top-k logical cache. In contrast, in the Twitter benchmark, it is the other way round. There, the top-k logical cache considers less segments than the top-k semantic cache.

8.5.2.10. Scale and Number of Used Segments

Finally, we investigate scale versus number of used segments for the cache size of 0.2 percent of scale (see figure 8.29).

*Expected behavior:* The number of used segments rises linearly in relation to scale (or cache size, respectively).

*Observed behavior and discussion:* The number of used segments of the top-k logical cache does rise linearly with scale (or cache size). Again, like in the Twitter benchmark, the number of used segments of the top-k semantic cache does not. On average, it uses just 2.3 segments (standard deviation: 1.9) to answer a query.
Figure 8.29: YCSB: Scale versus number of used segments for the cache size of 1 percent of scale
8.6. MonArch 3

The MonArch Digital Archiving System is a metadata repository for the management of digital documents with a spatial context [SWF11]. MonArch indexes documents according to their structural position in the building. In addition, documents can be assigned semantic properties that are organized in an ontology. Stored documents can be retrieved using the building structure, their semantic properties, any available textual information, or a combination thereof.

8.6.1. Description

Our colleague Alexander Stenzer has recorded and provided us with the queries issued by the MonArch 3 prototype during two typical user sessions. In these sessions, the user traverses through the different building parts of the tree-like structure of the building – called partonomy – until he or she reaches the building part that contains the desired documents. In addition, the user uses semantic properties to filter the documents. The first test set consists of 199 queries. The second set contains 212 queries. Listing 8.3 shows a typical test query. These queries usually join together multiple tables. And they contain where conditions composed of several conjunctions and disjunctions.

```
SELECT * 
FROM "Partonomy" AS "p", 
    "V_Partonomy_Hierarchy" AS "ph", 
    "R_Entities_located_at_Partonomy" AS "er" 
WHERE ( 
    ( 
    "er"."Partonomy_ID" = "ph"."Ancestor_ID" 
    AND "ph"."ID" = "p"."ID" 
    ) 
    ) 
    OR ( 
    "er"."Partonomy_ID" = "p"."ID" 
    AND "ph"."Ancestor_ID" = 1 
    AND "ph"."ID" = 2 
    ) 
    AND "p"."UID" = '72782d57-ae88-4821-86dc-90db20dc89d5' 
ORDER BY "p"."ID", "ph"."ID", "er"."ID"
```

Listing 8.3: Query $q_{monarch}$: a typical query of the test queries of MonArch 3

We were also able to obtain the corresponding database instance of the MonArch system that these queries were issued to. The database instance consists of various tables. In relative, the same 11 017 unique tuples were used by the test queries of both the first test set and the second test set.
8. Evaluation

8.6.2. Evaluation

In the evaluation, we will investigate the hit rates and relative query execution times of the two sets of test queries.

8.6.2.1. Cache Size and Hit Rate

To begin with, we take a look at cache size versus hit rate for the first test set (see figure 8.30) and the second test set (see figure 8.31).

![Figure 8.30: MonArch 3: Cache size versus hit rate for a first test set](image)

**Expected behavior**: If the cache size is increased, the hit rate will rise.

**Observed behavior and discussion**: In the first test set, the hit rate of the semantic and logical caches rises as the cache size is increased, until it reaches a plateau of about 70 percent, beginning at a cache size of about 800 tuples.

In the second test set, the hit rate of the semantic and logical caches also rises, but with much more unsteadiness. At about a cache size of 2000 tuples, it reaches a plateau of about 85 percent.

With both of the test sets, the course of the hit rate of the hash cache closely matches the hit rates of the semantic and logical caches. But the hash cache is only able to attain a slightly lower hit rate than the semantic and logical caches.

Especially interesting is the second test set with a cache size between 1000 and 1500 tuples. For these cache sizes, the semantic and logical caches can achieve a much higher hit rate than the hash cache. The reason for this observed behavior is that the second test set contains queries with very large results (more than 600 tuples) that take...
a lot of space in the cache. All caches can only fit in a small amount of elements this large. But the semantic and logical caches can use the available segments to answer queries that are not contained in the cache themselves. The hash cache cannot. It needs to have stored the results of the posed queries.

8.6.2.2. Cache Size and Execution Time

Next, we investigate cache size versus relative query execution time for the first test set (see figure 8.32) and the second test set (see figure 8.33).

Expected behavior: If the cache size is increased, the relative query execution time will fall until a plateau is reached.

Observed behavior and discussion: The observed behavior matches the expected one. The top-k semantic cache and the top-k logical cache have a slightly better performance than the hash cache. But the effect is very small. For the MonArch system, we recommend the integration of a hash cache, which is able to cut the query execution time in half.
8. Evaluation

Figure 8.32.: MonArch 3: Cache size versus relative query execution time for a first test set

Figure 8.33.: MonArch 3: Cache size versus relative query execution time for a second test set
8.7. Discussion and Comparison of Results

Overall, we were able obtain the following results:

**Viability**: In all benchmarks, top-k semantic caching was able to achieve lower, i.e., better, relative query execution times than the baseline hash cache. Hence, top-k semantic caching is indeed viable.

**Cache size**: To enable effective top-k semantic caching, a very small cache size is sufficient (Twitter: 0.2 percent of scale, YCSB: 1 percent of scale).

**Latency**: The server latency is not very relevant. Top-k semantic caching is very powerful for both low and high latencies.

**Throughput**: The top-k semantic cache will be extremely effective if the throughput is low, especially if the throughput is lower than 1000 kbps. In addition, we have observed that the lower the throughput, the higher the cache size should be.

**Pipelining**: Pipelining queries that can only be partially answered by the top-k semantic cache yields a significant advantage. Top-k semantic caching with pipelining that estimates a lower bound for the result of the remainder query does indeed outperform its counterpart without pipelining.

**Estimation performance**: The quality of the estimation of the lower bound of the result of the remainder query does not need to be very good. Even for an estimation performance of 50 percent, a significantly lower, i.e., better, relative query execution time can be achieved. Using targeted histograms, the number of buckets that are needed to achieve an estimation performance of 50 percent can be drastically reduced.

**Satisfiability checks**: The SMT solver must be able to solve expressions in less than about 1ms (Twitter) or about 5ms (YCSB) to allow the top-k semantic cache to achieve a significant improvement in comparison to the hash cache or no cache at all. Fortunately, the hybrid solver is able to check the expressions of all utilized benchmarks in far less than 1ms (see section 3.2.2).

**Semantic caching versus logical caching**: In all benchmarks, the top-k semantic cache achieved better results than the top-k logical cache. Generally, the top-k semantic cache generates more complex expressions for the SMT solver than the top-k logical cache. But thereby, it gains that each tuple is contained in the cache only once, which usually leads to higher hit rates. In addition, we have observed that the top-k semantic cache only needs a constantly small amount of segments to answer a query, while the number of used segments of the top-k logical cache rises linearly with scale and cache size. For the top-k logical cache, this leads to a significant performance loss (see YCSB in section 8.5 especially).
8. Evaluation

In conclusion, the top-k semantic cache is an effective cache for low throughputs that can achieve very good results for extremely small cache sizes and that successfully enables pipelining of top-k queries.
9. Conclusion

In this chapter, we discuss future work and summarize the main aspects, findings and lessons learned about top-k semantic caching in this thesis.

9.1. Future Work

The top-k semantic cache could be enhanced by adding prefetching and context-awareness.

9.1.1. Prefetching

A natural extension of the top-k semantic cache is the introduction of prefetching. Thereby, the cache could be loaded with query results when the connection has a higher bandwidth to enable accelerated query answering in situations with high latency, low throughput and connection disruptions [Dra05]. In addition, the limited battery life of mobile devices can be extended by prefetching data nightly, when the phone is charging. [LRS12]

In his 2013 master's thesis “Prefetching for Key-Value Stores” [Hub13], Pius Hübl surveyed and classified various prefetching techniques. He studied rule-based prefetching techniques like CloSpan [YHA03], C-Miner [LCSZ04] and QuickMine [SMA08]. And he looked at layout-based prefetching algorithms like ReadAhead [FHC08] and DiskSeen [DJC07]. In addition, he researched graph-based prefetching techniques like dependency graph [GA94], markov predictor [JG97], PPM (Prediction by Partial Matching) [BGJ07] and WMO (Web log Mining with Ordering) [NM01, NKM03]. He has implemented the two most promising ones, QuickMine and WMO, for a key-value store at the server. He evaluated them in different scenarios using an LRU cache as a base line. In conclusion, he recommended the prefetching algorithm WMO, which could achieve the best results overall.

In his 2014 master's thesis “Prefetching for Databaseses” [Gan14], Stefan Ganser adapted the WMO prefetching scheme [NM01, NKM03] for client-side prefetching of parameterized SQL queries. To realize this adaption, he has utilized techniques from C-Miner [LCSZ04] and Scalpel [Bow05, BS05, IB06, BS07]. His implementation heavily builds on the IQCache framework, especially the component that can parse SQL queries and attribute query meta data. It does not use the top-k semantic cache, but rather uses a very simple LRU cache that can only answer requests if the cache contains an
9. Conclusion

exact match. He evaluated the adapted WMO prefetching scheme using the TPC-C benchmark [LD93, Raa10, RKS] and the AuctionMark benchmark [AP, APSZ12]. The introduction of the prefetching scheme has been able to significantly reduce the total execution time of the TPC-C benchmark and the AuctionMark benchmark. Based on these very promising results, we expect that prefetching can be successfully applied to the top-k semantic cache to increase performance and versatility. The WMO prefetching scheme is a prime candidate to enable prefetching. Though, further adaption of the WMO prefetching scheme and several modifications to the top-k cache are certainly still required. For example, the top-k semantic cache currently does not support parameterized queries. Only the component that can parse SQL queries and attribute query meta data does. Then, the top-k semantic cache would not only be able to cache queries, but would have the power of prefetching at its disposal.

9.1.2. Context

Context information is an important reason why humans are so successful in conveying information. They have an understanding how the world works, how daily events play out. They are able to use implicit information. Thereby, context information can dramatically increase the information bandwidth of the conversation. For example, if someone says, “I really like this book!”, the said sentence only contains the information that someone really likes some book. But we need context information to know who really likes which book. If we know that the person talking to us is actually our good friend Alexander, who is holding the book “The Hitchhiker's Guide to the Galaxy” in his hands, the sentence actually conveys the information that Alexander really likes the book “The Hitchhiker's Guide to the Galaxy”.

Fundamentally, context has three important aspects. First, it is situation-dependent. Secondly, its relevance is dependent on the subject of the interaction. Thirdly, context is additional information that is not directly conveyed in the conversation. Together, the definitions of context by Abowd et al. and Lieberman und Selker characterize context quite well:

“Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and application, including the user and application themselves.” [ADB+99]

“Context can be considered to be everything that effects the computation except explicit input and output.” [LS00]

Systems that utilize context and adapt its behavior dependent on context are called context-aware:
9.1. Future Work

“A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user’s task.”

Context can be used to enhance caches, because the context information may be used to intelligently decide which items to cache using context-aware cache replacement strategies. In addition, context will enable intelligent prefetching if context information provides conclusions about items that will be needed in the future.

An obvious type of context is the location. And, in fact, location-based caching and prefetching has been very prevalent in research for quite some time [Bli95, Nel98, Dra06, IM+10, LBC+11, LFS13]. But there are also more recent approaches to caching and prefetching that use general context information [vB04, BCQ+07, Bun08, MKB+08, BGMC13, BDFMWB13]. Furthermore, we have researched context-aware prefetching in a semantic cache, albeit not a top-k semantic cache [Ehl09].

Adding context-aware caching and prefetching techniques to the top-k semantic cache seems very promising.

9.1.3. The Mobile Use Case

The top-k semantic cache is aimed at a situation with low throughput and high latency. The mobile use case is such a scenario. Especially in rural areas, mobile devices can suffer from these problems. But the mobile use case does involve some challenges that have to be dealt with.

First of all, applications on mobile devices usually do not utilize a JDBC database connection. They do not use SQL. But as long as we are able to describe the cached sets of elements by logical expressions, the techniques of this thesis can be applied. The top-k semantic cache can be implemented as a library that is a component of the mobile application.

Secondly, in our prototype IQCache, which is written in Java, we used SMT solvers like Yices [DM06] and MathSat 5 [CGSS13] that are not implemented in Java. They were integrated into IQCache via the Java Native Interface. Hence, the prototype might be relatively easily ported to a mobile Android device with the Android Development Tools and the Android SDK [Goo14]. But the SMT solvers might not, since we are missing a fast solver that is implemented in Java. Fortunately, it is possible to port native libraries to an Android system using the Android Native Development Kit [KC14].

To port IQCache to an iOS system, the top-k semantic cache would possibly have to be re-implemented in Objective-C or Swift [App, App14]. Or alternatively, a Java-to-Objective-C code bridge like RoboVM could be used [Kri14, Tri].

Furthermore, mobile devices have slow processors in comparison with desktop computers. We use GeekBench 3 [Pri14], a cross-platform processor benchmark, to compare processing speeds. In this benchmark, current mobile devices like the Apple iPhone 6,
9. Conclusion

the Samsung S5, the Samsung S6 Edge, and the HTC One M9 achieve a significantly lower score than current desktop computers (see table 9.1).

<table>
<thead>
<tr>
<th>Processor / Mobile device</th>
<th>GeekBench 3 score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7 4810MQ 2.8GHz</td>
<td>3062 [Pri]</td>
</tr>
<tr>
<td>Intel Core i5 4300M 2.6GHz</td>
<td>2934 [Pri]</td>
</tr>
<tr>
<td>Intel Core 2 Quad Q9400 2.66GHz</td>
<td>2431 [Pri]</td>
</tr>
<tr>
<td>Apple iPhone 6 (Apple A8)</td>
<td>1628 [Klu14]</td>
</tr>
<tr>
<td>Samsung S6 Edge (Exynos 7420)</td>
<td>1505 [Hry15]</td>
</tr>
<tr>
<td>HTC One M9 (Qualcomm Snapdragon 810)</td>
<td>1200 [Hry15]</td>
</tr>
<tr>
<td>Samsung S5 (Qualcomm Snapdragon 801)</td>
<td>947 [Klu14]</td>
</tr>
</tbody>
</table>

Table 9.1.: Comparison of GeekBench 3 results of processors and mobile devices

To assess the applicability of top-k semantic caching to current mobile devices, we have used the Twitter benchmark [CHBGTD], which is particularly demanding on the utilized SMT solver. We have underclocked the Intel Core i7 4810MQ 2.8GHz and Intel Core i5 4300M 2.6GHz to 90, 80, ..., 30 percent processing power. (The Intel Core 2 Quad Q9400 2.66GHz could unfortunately not be underclocked in the same way.) We have plotted the GeekBench 3 scores versus the average satisfiability check duration that the processor can achieve using the expressions that are issued to the SMT solver during the Twitter benchmark under the constraints (see figure 9.1). We assume a minimum needed performance of 1ms per satisfiability check (see section 8.4). Based on this results, we can conclude that the Samsung S5 is probably too slow to support top-k semantic caching. In contrast, the Apple iPhone 6 may just allow for it. And in future mobile devices, which can only be expected to have more powerful processors, top-k semantic caching will certainly be possible.
9.1. Future Work

![Figure 9.1: GeekBench 3 score versus average satisfiability check duration](image)

Figure 9.1.: GeekBench 3 score versus average satisfiability check duration
9. Conclusion

9.2. Conclusion

In this thesis, we have presented an innovative semantic cache that naturally supports top-k queries. After a motivating introduction, we have introduced the running example, which has kindly accompanied us through this thesis. Then, we have discussed our assumptions and contributions. In addition, we have extensively surveyed the related work of semantic caching. Afterward, we have introduced SMT solvers and our novel hybrid solver. This hybrid solver calculates the size of the disjunctive normal form of an expression and uses this to assess the complexity of the expression. Using the novel formula, the hybrid solver is able to distinctly outperform all other considered solvers. We have discussed the definition and description of segments, the cache elements of the top-k semantic cache, in great detail. We have presented the algorithms for set operations like difference, intersection, and union on segments, which the top-k semantic cache needs for cache management. Furthermore, we have presented a new algorithm that can estimate the lower bounds of results of sorted queries using multidimensional histograms. With this algorithm, our top-k semantic cache is able to pipeline results of top-k queries. To coherently exemplify all new algorithms for cache management as well as pipelining, we have used the running example. We have introduced a new and innovative algorithm for top-k query processing that uses stored top-k query results to answer given top-k queries. We have implemented a prototype of a top-k semantic cache called IQCache (Intelligent Query Cache). An extensive and thorough evaluation of our prototype IQCache using the benchmarks Yahoo! Cloud Serving Benchmark with a ScanRecord workload, the Twitter benchmark with a GetTweetsFromFollowing workload, and the MonArch 3 test queries has demonstrated the viability of top-k semantic caching in practice. Our experiments have revealed that the top-k semantic cache is an effective cache for low throughputs, e.g., less than 1024 kbps. It achieves very good results for very small cache sizes, e.g., less than 1 percent of the tuples in the database instance. The top-k semantic cache was able to consistently outperform hash-based caching strategies. Furthermore, the evaluation has also shown that the top-k semantic cache successfully uses pipelining to speed up the execution time of top-k queries. Finally, top-k semantic caching still offers exciting research opportunities. We could introduce prefetching. We could add context-awareness. Or we could really delve into the mobile use case and study the challenges of the adoption of top-k semantic caching to mobile devices.
A. Histogram Creation with Window Functions

Multidimensional equi-depth histograms can be heuristically generated by a single SQL query using window functions, which are available in SQL since SQL:2003 [ISO03]. Assume that we use an instance of the relation post from the running example (see section 1.2) with 250 000 tuples. To represent this relation, we want to create a histogram on the dimensions id and likes with a maximum of 2 500 buckets. First, we have to divide each dimension into intervals. For example, IQCache chooses to divide the dimension id into 59 intervals and the dimension likes into 42 intervals, because there are more different ids than there are different likes. Because 59 · 42 = 2 478 < 2 500 holds, IQCache will create a maximum of 2 478 buckets, which is less than the allowed maximum of 2 500 buckets.

Secondly, IQCache issues the query shown in listing A.1 that creates the buckets of the histogram on the dimensions id and likes for the relation post. The execution of this query on the instance with 250 000 tuples takes less than 3 seconds.

```sql
SELECT b0.bid AS bid0, b0.blower AS blower0, b0.bhigher AS bhigher0,
       b1.bid AS bid1, b1.blower AS blower1, b1.bhigher AS bhigher1,
       COUNT(*) AS bcount
FROM post t0,
     (SELECT bid, MIN(id) AS blower, MAX(id) AS bhigher
      FROM    SELECT id, (CAST(FLOOR((
                                       (CAST(((RANK() OVER (ORDER BY id)) - 1) AS DOUBLE PRECISION))
                                     ) / (SELECT COUNT(id) AS counta
                                     FROM post
                                   ) ) * (SELECT GREATEST(0, numberofbucketsn - (SELECT DISTINCT 1 AS notnull
                                   FROM post
                                 WHERE id IS NULL
                             UNION ALL
                           SELECT DISTINCT 0
                           FROM post
                         ) )
```
A. Histogram Creation with Window Functions

```sql
FROM post
WHERE id IS NOT NULL AND NOT EXISTS(
    SELECT *
    FROM post
    WHERE id IS NULL
)
) AS numberofbuckets
FROM (
    SELECT LEAST(GREATEST(1, 59), countdn) AS numberofbucketsn
    FROM (SELECT COUNT(*) AS countdn
        FROM (SELECT DISTINCT id
            FROM post
        ) t
    )
        ) AS INTEGER) + 1) AS bid
FROM post
WHERE id IS NOT NULL
UNION ALL
SELECT NULL, (
    SELECT LEAST(GREATEST(1, 59), countdn) AS numberofbucketsn
    FROM (SELECT COUNT(*) AS countdn
        FROM (SELECT DISTINCT id
            FROM post
        ) t
    )
        ) AS INTEGER) + 1) AS bid
FROM post
WHERE id IS NULL
GROUP BY bid
)
) b0,

( SELECT bid, MIN(likes) AS blower, MAX(likes) AS bhigher
FROM ( SELECT likes, (CAST(FLOOR((
    CAST(((RANK() OVER (ORDER BY likes)) - 1) AS DOUBLE PRECISION))
    ) / (SELECT COUNT(likes) AS counta
        FROM post ) * (SELECT GREATEST(
            0, numberofbucketsn - (SELECT DISTINCT 1 AS notnull
```
FROM post
WHERE likes IS NULL
UNION ALL
SELECT DISTINCT 0
FROM post
WHERE likes IS NOT NULL AND NOT EXISTS(
  SELECT *
  FROM post
  WHERE likes IS NULL
)
)
) AS numberofbuckets
FROM (SELECT LEAST(GREATEST(1, 42), countdn) AS numberofbucketsn
FROM (SELECT COUNT(*) AS countdn
FROM (SELECT DISTINCT likes
FROM post
)
) t
)
) t
) AS INTEGER) + 1) AS bid
FROM post
WHERE likes IS NOT NULL
UNION ALL
SELECT NULL, (SELECT LEAST(GREATEST(1, 42), countdn) AS numberofbucketsn
FROM (SELECT COUNT(*) AS countdn
FROM (SELECT DISTINCT likes
FROM post
)
) t
)
) t
FROM post
WHERE likes IS NULL
) t
GROUP BY bid
WHERE
(b0.blower <= t0.id AND t0.id <= b0.bhigher)
OR
(b0.bhigher IS NULL AND t0.id IS NULL )
AND
(b1.blower <= t0.likes AND t0.likes <= b1.bhigher)
A. Histogram Creation with Window Functions

Listing A.1: Query that creates buckets of the histogram on the dimensions id and likes for the relation post
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography

[119x747]


Bibliography


237
Bibliography


Bibliography


[WHZL04] Hai Wan, Xiao-Wei Hao, Tao Zhang, and Lei Li. Semantic caching services for data grids. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun,


[YHA03] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: mining closed sequential patterns in large datasets. In *Proc. SDM ’03*, pages 166–177, 2003.


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Semantic caching with probe query and remainder query</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Entity-relationship model of the running example using Chen’s notation [Che76]</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Table definitions of the running example</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Overlap between query $q_1$ and query $q_2$</td>
<td>9</td>
</tr>
<tr>
<td>1.5</td>
<td>Pipelining possibility of query $q_2$</td>
<td>10</td>
</tr>
<tr>
<td>1.6</td>
<td>Overlap between query $q_1$ and query $q_3$</td>
<td>10</td>
</tr>
<tr>
<td>1.7</td>
<td>Overlap between query $q_2$ and query $q_3$</td>
<td>11</td>
</tr>
<tr>
<td>1.8</td>
<td>Pipelining possibility of query $q_3$</td>
<td>12</td>
</tr>
<tr>
<td>1.9</td>
<td>Overlaps of query $q_1$, query $q_2$ and query $q_3$</td>
<td>12</td>
</tr>
<tr>
<td>1.10</td>
<td>Set-up with high latency and low throughput</td>
<td>14</td>
</tr>
<tr>
<td>1.11</td>
<td>Set-up with top-k semantic cache</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Performance of the hybrid solver in comparison with Yices, MathSAT 5 and Z3 for expressions of the prototype IQCache during Yahoo! Cloud Serving Benchmark and Twitter benchmark</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Overlap of two queries (e.g., query $q_1$ overlaps with query $q_2$)</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Subsumption of two queries (e.g., query $q_2$ subsumes query $q_3$)</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Segment $s_1$ for query $q_1$</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Conditions of a segment of a top-k semantic cache</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Conditions of a segment of a top-k semantic cache with implicitly defined i</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Creating a segment $s_2$ for query $q_2$ using segment $s_1$ of query $q_1$</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Segment $s_2$ for query $q_2$</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Difference of two segments</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Segment $s_1$ after creation and subtraction of segment $s_2$</td>
<td>80</td>
</tr>
<tr>
<td>4.8</td>
<td>Creating a segment $s_3$ for query $q_3$ using segment $s_2$ of query $q_1$</td>
<td>85</td>
</tr>
<tr>
<td>4.9</td>
<td>Segment $s_3$ for query $q_1$</td>
<td>88</td>
</tr>
<tr>
<td>4.10</td>
<td>Segment $s_2$ after creation and subtraction of segment $s_3$</td>
<td>90</td>
</tr>
<tr>
<td>4.11</td>
<td>Intersection between two segments of the top-k semantic cache</td>
<td>92</td>
</tr>
<tr>
<td>4.12</td>
<td>Union of two segments of the top-k semantic cache</td>
<td>96</td>
</tr>
<tr>
<td>5.1</td>
<td>A sample histogram for the tuples from the running example</td>
<td>121</td>
</tr>
</tbody>
</table>
List of Figures

5.2. Histogram after execution of query $q_1$ ................................................. 122
5.3. Histogram before processing of query $q_2$ ........................................... 133
5.4. Estimated bound of remainder query of query $q_2$ ................................. 136
5.5. Effect of pipelining of query $q_2$ ............................................................ 136
5.6. Histogram after execution of query $q_2$ .................................................. 137
5.7. Histogram before execution of query $q_3$ .................................................. 139
5.8. Estimated bound of remainder query of query $q_3$ ................................. 143
5.9. Example for a skyline of buckets of a query $q$ with an ordering based on a function $f$ from example 31 in a grid structured multidimensional histogram based on the attributes $likes$ and $dislikes$ .................................................. 145
6.1. Graph of the foreign key constraints of the tables of the running example ... 150
6.2. A top-k semantic cache answers a query .................................................. 156
7.1. Overview of components of IQCache ....................................................... 167
8.1. Twitter: Cache size versus hit rate for the scale of 400 000 tuples ............. 180
8.2. Twitter: Cache size versus relative query execution time for the scale of 400 000 tuples ................................................................. 181
8.3. Twitter: Server latency versus relative query execution time for cache size of 800 tuples and the scale of 400 000 tuples .................................................. 181
8.4. Twitter: Throughput versus relative query execution time for cache size of 800 tuples and the scale of 400 000 tuples .................................................. 182
8.5. Twitter: Average duration of a single satisfiability check versus relative query execution time for cache size of 800 tuples and the scale of 400 000 tuples ................................................................. 183
8.6. Twitter: Estimation performance versus relative query execution time for cache size of 800 tuples and the scale of 400 000 tuples (considering only pipelined queries) .................................................. 184
8.7. Twitter: Number of buckets versus estimation performance for cache size of 800 tuples and the scale of 400 000 tuples (full histogram) ............ 185
8.8. Twitter: Number of buckets versus estimation performance for cache size of 800 tuples and the scale of 400 000 tuples (targeted histogram) .... 185
8.9. Twitter: Scale versus hit rate for the cache size of 0.2 percent of scale ....... 186
8.10. Twitter: Scale versus relative query exection time for the cache size of 0.2 percent of scale (top-k cache types) .................................................. 187
8.11. Twitter: Scale versus relative query exection time for the cache size of 0.2 percent of scale (all cache types) .................................................. 187
8.12. Twitter: Scale versus number of considered segments for the cache size of 0.2 percent of scale ................................................................. 188

244
List of Figures

8.13. Twitter: Scale versus number of used segments for the cache size of 0.2 percent of scale ................................................. 189
8.14. YCSB: Cache size versus hit rate for the scale of 800,000 tuples ............... 191
8.15. YCSB: Cache size versus relative query execution time for the scale of 800,000 tuples .................................................. 192
8.16. YCSB: Server latency versus relative query execution time for the cache size of 8,000 tuples and the scale of 800,000 tuples .................. 192
8.17. YCSB: Throughput versus relative query execution time for the cache size of 8,000 tuples and the scale of 800,000 tuples .......... 193
8.18. YCSB: Throughput versus relative query execution time for the cache size of 40,000 tuples and the scale of 800,000 tuples ............. 194
8.19. YCSB: Average duration of a single satisfiability check versus relative query execution time for the cache size of 8,000 tuples and the scale of 800,000 tuples ............................... 194
8.20. YCSB: Estimation performance versus relative query execution time for the cache size of 8,000 tuples and the scale of 800,000 tuples (considering only pipelined queries) ............................. 195
8.21. YCSB: Number of buckets versus estimation performance for cache size of 8,000 tuples and the scale of 800,000 tuples (full histogram) ........ 196
8.22. YCSB: Scale versus estimation performance for cache size of 1 percent of scale and for number of buckets of 1 percent of scale (full histogram) ................................................. 197
8.23. YCSB: Number of buckets versus estimation performance for cache size of 8,000 tuples and the scale of 800,000 tuples (targeted histogram) .... 198
8.24. YCSB: Scale versus estimation performance for cache size of 1 percent of scale and for number of buckets of 0.05 percent of scale (targeted histogram) ................................................ 198
8.25. YCSB: Scale versus hit rate for the cache size of 1 percent of scale ............. 199
8.26. YCSB: Scale versus relative query execution time for the cache size of 1 percent of scale (top-k cache types) ......................... 200
8.27. YCSB: Scale versus relative query execution time for the cache size of 1 percent of scale (all cache types) .................................... 200
8.28. YCSB: Scale versus number of considered segments for the cache size of 1 percent of scale .................................................. 201
8.29. YCSB: Scale versus number of used segments for the cache size of 1 percent of scale .......................................................... 202
8.30. MonArch 3: Cache size versus hit rate for a first test set ......................... 204
8.31. MonArch 3: Cache size versus hit rate for a second test set ................... 205
8.32. MonArch 3: Cache size versus relative query execution time for a first test set ................................................................. 206
8.33. MonArch 3: Cache size versus relative query execution time for a second test set ................................................................. 206
List of Figures

9.1. GeekBench 3 score versus average satisfiability check duration . . . . . . 213
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Sample instance of the relation <em>post</em> of the running example</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of domains</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Conjunction in three-valued logic (i.e., ∧)</td>
<td>41</td>
</tr>
<tr>
<td>3.7</td>
<td>Disjunction in three-valued logic (i.e., ∨)</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Negation in three-valued logic (i.e., ¬)</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td><em>Isunknown</em> in three-valued logic (i.e., u)</td>
<td>42</td>
</tr>
<tr>
<td>3.10</td>
<td>Truth table that proves ( u(\neg \hat{a}) \equiv \hat{u} )</td>
<td>42</td>
</tr>
<tr>
<td>3.11</td>
<td>Truth table that proves ( u(\hat{a} \land \hat{b}) \equiv (\hat{u} \land \hat{b}) \lor (\hat{u} \land \hat{b}) )</td>
<td>43</td>
</tr>
<tr>
<td>3.12</td>
<td>Expressions ( \hat{p}_1 ) and ( \hat{p}_2 ), an example for partial equivalence</td>
<td>45</td>
</tr>
<tr>
<td>3.13</td>
<td>The function ( f_{\text{true}} ) converts expressions in three-valued logic to expressions in two-valued logic</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>Values of query ( q_1 )</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview of parts of a top-k semantic cache segment</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Query result of query ( q_1 ) using sample instance of running example</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Values of segment ( s_1 )</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Values of empty segment ( s'_2 ) created by segmentCreate(( q_2 )) (see algorithm 3)</td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Result of ( \pi_{A_{\text{select}}} \mathcal{S}_{\mathcal{T}_2} (\mathcal{T}_1) )</td>
<td>71</td>
</tr>
<tr>
<td>4.7</td>
<td>Values of segment ( s_2 ) after the insertion of tuples from overlaps</td>
<td>74</td>
</tr>
<tr>
<td>4.8</td>
<td>Query execution plans of ( r_2 ) and ( r'_2 ) in PostgreSQL</td>
<td>75</td>
</tr>
<tr>
<td>4.9</td>
<td>Result of ( r_2 := q_2 \setminus (\pi_{A_{\text{select}}} \mathcal{S}_{\mathcal{T}_2} (\mathcal{T}_1)) )</td>
<td>75</td>
</tr>
<tr>
<td>4.10</td>
<td>Values of segment ( s_2 ) after completed processing of query ( q_2 )</td>
<td>76</td>
</tr>
<tr>
<td>4.11</td>
<td>Values of segment ( s_1 ) after subtraction of segment ( s_2 )</td>
<td>81</td>
</tr>
<tr>
<td>4.12</td>
<td>Values of empty segment ( s_3 ) created by segmentCreate(( q_3 )) (see algorithm 3)</td>
<td>82</td>
</tr>
<tr>
<td>4.13</td>
<td>Result of ( \pi_{A_{\text{select}}} \mathcal{S}_{\mathcal{T}_3} (\mathcal{T}_1) )</td>
<td>84</td>
</tr>
<tr>
<td>4.14</td>
<td>Values of segment ( s_3 ) after the insertion of tuples from overlaps</td>
<td>86</td>
</tr>
<tr>
<td>4.15</td>
<td>Result of ( r_3 := q_3 \setminus (\pi_{A_{\text{select}}} \mathcal{S}_{\mathcal{T}_3} (\mathcal{T}_1)) )</td>
<td>86</td>
</tr>
<tr>
<td>4.16</td>
<td>Values of segment ( s_3 ) after completed processing of query ( q_3 )</td>
<td>87</td>
</tr>
<tr>
<td>4.17</td>
<td>Values of segment ( s_2 ) after subtraction of segment ( s_3 )</td>
<td>91</td>
</tr>
<tr>
<td>4.18</td>
<td>Result of query ( \mathcal{S}_{\mathcal{T}_3} (\mathcal{T}_1) )</td>
<td>94</td>
</tr>
<tr>
<td>4.19</td>
<td>Values of segment ( s_1 )</td>
<td>95</td>
</tr>
<tr>
<td>4.20</td>
<td>Values of segment ( s_3 )</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.24</td>
<td>Values of segment $s_{u*} := s_1 \cup^* s_3$</td>
<td>105</td>
</tr>
<tr>
<td>5.1</td>
<td>Employed histogram types in commercial and free database systems</td>
<td>107</td>
</tr>
<tr>
<td>5.7</td>
<td>Tuples of the overlap of query $q_2$ and segment $s_1$</td>
<td>135</td>
</tr>
<tr>
<td>5.10</td>
<td>Tuples of the overlap of query $q_3$ and segment $s_2$</td>
<td>143</td>
</tr>
<tr>
<td>6.1</td>
<td>Possible joins of running example with two and three tables</td>
<td>150</td>
</tr>
<tr>
<td>7.1</td>
<td>Different eclipse projects of the IQCache workspace with source lines of</td>
<td>169</td>
</tr>
<tr>
<td></td>
<td>code (SLOC) and comment lines of code (CLOC)</td>
<td></td>
</tr>
<tr>
<td>8.2</td>
<td>Utilized values of the different domains</td>
<td>175</td>
</tr>
<tr>
<td>8.3</td>
<td>Used scales of the benchmarks</td>
<td>176</td>
</tr>
<tr>
<td>8.4</td>
<td>Used cache sizes of the benchmarks</td>
<td>176</td>
</tr>
<tr>
<td>8.5</td>
<td>Utilized values of the analyzed domains</td>
<td>177</td>
</tr>
<tr>
<td>8.6</td>
<td>Overview of analyses of the evaluation</td>
<td>178</td>
</tr>
<tr>
<td>9.1</td>
<td>Comparison of GeekBench 3 results of processors and mobile devices</td>
<td>212</td>
</tr>
</tbody>
</table>
# List of Algorithms

1. `dnfLength(\hat{e})` ......................................................... 34

2. `\hat{e} \leq \lambda`: `conditionOrderBy(O, \lambda)` ......................... 62

3. `segmentCreate(q)` .......................................................... 68

4. `segmentInitialInsert(s_{new}, \{s_1, \ldots, s_n\})` .................... 68

5. `segmentComplete(s)` ....................................................... 69

6. `s_c \setminus s_{new}`: `segmentDifferenceInPlace(s_c, s_{new})` .......................... 78

7. `s_c \cap s_{new}`: `segmentIntersection(s_c, s_{new})` .................. 93

8. `s_1 \cup s_2`: `segmentUnionInPlace(s_1, s_2)` ....................... 97

9. `s_1 \cup^* s_2`: `segmentUnionInPlace^*(s_1, s_2)` ....................... 102

10. `getLowerBound(H^e, q, \tau)` — idea (without index usage) .......................... 117

11. `getLowerBound(b, O)` ...................................................... 118

12. `overlapsBucket(\hat{q}, b)` ............................................. 119

13. `chooseTuples(b)` .............................................................. 119

14. `getLowerBound(H^e, q, \tau)` ........................................... 123

15. `getSmallestBucket(H^e, q, \tau, i, r)` ..................................... 124

16. `exitCondition(H^e, q, \tau, r)` ........................................... 125

17. `scanRegion(H^e, \tau, r)` ................................................. 125

18. `chooseRegions(H^e, q, i, r)` ............................................ 126

19. `splitRegion(r, a)` .......................................................... 127

20. `processRegions(H^e, q, \tau, j, S)` ....................................... 128

21. `overlapsRegion(\hat{q}, r)` .............................................. 129

22. `selectBucket(H^e, q, \tau, r)` ........................................... 130

23. `getLowerBound(H^e, q, \tau)` — for orderings based on component-wise monotonous functions .................................. 146

24. `getJoins(q)` ............................................................... 151

25. `getEqualAttributes(\hat{e})` ................................................ 151

26. `cacheOverlappingSegments(Cache_{R,J}, q)` ............................ 153

27. `segmentLoadAttributes(s, A^{new})` ........................................... 154

28. `segmentLoadTuples(s, l)` ................................................. 155

29. `cacheAnswerQuery(Cache, q)` ........................................... 157

30. `cacheAddFullCoal(Cache_{R,J}, q, L, T, t)` ................................ 160
List of Algorithms

31. cacheAddNoCoal(Cache_{R,J}, q, L, T_r, l) .......................... 163
# List of Listings

1. Query $q_1$: a top-k query returning last week’s most liked posts .......................... 6
2. Query $q_2$: a top-k query returning today’s posts by id ................................. 7
3. Query $q_3$: a top-k query returning the most liked posts of the last hour ........ 7

3.1. Query $p_1$ ........................................................................................................................................ 44
3.2. Query $p_2$ ........................................................................................................................................ 44

4.1. Query $q_1$: a top-k query returning last week’s most liked posts .......................... 53
4.2. Query $q_2$: a top-k query returning today’s posts by id ................................. 70
4.3. Query $r_2$: remainder query for segment $s_2$ ......................................................... 72
4.4. Query $r_2^r$: rewritten remainder query for segment $s_2$ ............................... 74
4.5. Query $q_3$: a top-k query returning the most liked posts of the last hour ........ 81
4.6. Query $r_3$: remainder query for segment $s_3$ ......................................................... 84

6.1. Query $q_j$: a top-k query with joins ............................................................................. 152
6.2. Query $q_{\text{content}}$: a query that loads the missing attribute $\text{content}$ ...................... 154

8.1. Query $q_{\text{twitter}}$: a typical query of the Twitter benchmark ................................. 179
8.2. Query $q_{\text{ycsb}}$: a typical query of the Yahoo! Cloud Serving Benchmark .......... 190
8.3. Query $q_{\text{monarch}}$: a typical query of the test queries of MonArch 3 .................. 203

A.1. Query that creates buckets of the histogram on the dimensions $id$ and $likes$ for the relation $\text{post}$ .................................................................................................................. 217
<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Variable and Domain</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Varchar Domain</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>Size of Varchar Domain</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Lexicographical Order on Vchars</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Boolean satisfiability problem</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>Satisfiability modulo theories</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Example for dnfLength (see algorithm 1)</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>Partial equivalence</td>
<td>44</td>
</tr>
<tr>
<td>9</td>
<td>Partial equivalence – continued</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>Description of query q1</td>
<td>53</td>
</tr>
<tr>
<td>11</td>
<td>Segment s1 for query q1</td>
<td>55</td>
</tr>
<tr>
<td>12</td>
<td>Order by of segment s1</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>Order by condition (\hat{o}_1) of segment s1</td>
<td>63</td>
</tr>
<tr>
<td>14</td>
<td>Attribute score function</td>
<td>64</td>
</tr>
<tr>
<td>15</td>
<td>Score function</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>Order by condition with arbitrary score function</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>Segment s2 for query q2</td>
<td>70</td>
</tr>
<tr>
<td>18</td>
<td>Difference of segment s1 and segment s2</td>
<td>79</td>
</tr>
<tr>
<td>19</td>
<td>Segment s3 for query q3</td>
<td>81</td>
</tr>
<tr>
<td>20</td>
<td>Difference of segment s2 and segment s3</td>
<td>87</td>
</tr>
<tr>
<td>21</td>
<td>Intersection of segment s1 and segment s2</td>
<td>93</td>
</tr>
<tr>
<td>22</td>
<td>Union of segment s1 and segment s2</td>
<td>99</td>
</tr>
<tr>
<td>23</td>
<td>Union* of segment s1 and segment s3</td>
<td>102</td>
</tr>
<tr>
<td>24</td>
<td>Tuple counts</td>
<td>108</td>
</tr>
<tr>
<td>25</td>
<td>Lower bound of bucket b5 for order by (O_1)</td>
<td>118</td>
</tr>
<tr>
<td>26</td>
<td>Overlap of bucket (b_2) and expression (\hat{q}_1)</td>
<td>119</td>
</tr>
<tr>
<td>27</td>
<td>Sample multidimensional histogram</td>
<td>120</td>
</tr>
<tr>
<td>28</td>
<td>Updated histogram after execution of query q1</td>
<td>121</td>
</tr>
<tr>
<td>29</td>
<td>Lower bound for remainder query (r_2) of query q2</td>
<td>130</td>
</tr>
<tr>
<td>30</td>
<td>Lower bound for remainder query (r_2) of query q3</td>
<td>138</td>
</tr>
<tr>
<td>31</td>
<td>Component-wise monotony</td>
<td>144</td>
</tr>
<tr>
<td>32</td>
<td>Skyline of smallest buckets</td>
<td>145</td>
</tr>
<tr>
<td>33</td>
<td>Equivalence classes in running example</td>
<td>150</td>
</tr>
<tr>
<td>34</td>
<td>Joins of a query</td>
<td>152</td>
</tr>
<tr>
<td>Example</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>35</td>
<td>Loading of additional columns for segment $s_1$</td>
<td>154</td>
</tr>
<tr>
<td>36</td>
<td>Full coalescing of segment $s_2$</td>
<td>161</td>
</tr>
<tr>
<td>37</td>
<td>Full coalescing of segment $s_3$</td>
<td>161</td>
</tr>
<tr>
<td>38</td>
<td>No coalescing of segment $s_2$</td>
<td>164</td>
</tr>
</tbody>
</table>
List of Definitions

Definition 1. Variable .......................................................... 28
Definition 2. Domain function ............................................... 28
Definition 3. String Domain, Varchar Domain ......................... 28
Definition 4. Lexicographical Order on Varchars ...................... 29
Definition 5. Partial equivalence .......................................... 45
Definition 6. Overlap .......................................................... 46
Definition 7. Subsumption .................................................... 48
Definition 8. Query ............................................................. 53
Definition 9. Segment of a top-k semantic cache ...................... 54
Definition 10. Order by condition ......................................... 58
Definition 11. Order by component ....................................... 61
Definition 12. Order by ....................................................... 61
Definition 13. Attribute score function .................................. 64
Definition 14. Combining function ........................................ 64
Definition 15. Score function ............................................... 64
Definition 16. Full coalescing .............................................. 67
Definition 17. No coalescing ............................................... 67
Definition 18. Total union .................................................... 67
Definition 19. Multidimensional histogram ............................... 108
Definition 20. Bucket .......................................................... 108
Definition 21. Histogram for the estimation of query bounds ....... 109
Definition 22. Usability of a histogram for a specific query ....... 109
Definition 23. Inclusion between histograms .......................... 110
Definition 24. Histogram selection (HS) ............................... 112
Definition 25. 0-1 knapsack ............................................... 112
Definition 26. Component-wise monotony ............................. 144
# List of Propositions, Theorems and Corollaries

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition 1</td>
<td>Size of Varchar Domain</td>
<td>29</td>
</tr>
<tr>
<td>Proposition 2</td>
<td>Correctness of algorithm dnfLength</td>
<td>35</td>
</tr>
<tr>
<td>Proposition 3</td>
<td>Unknown and negation</td>
<td>42</td>
</tr>
<tr>
<td>Proposition 4</td>
<td>isunknown and conjunction</td>
<td>42</td>
</tr>
<tr>
<td>Proposition 5</td>
<td>isunknown and disjunction</td>
<td>42</td>
</tr>
<tr>
<td>Proposition 6</td>
<td>Composition of isunknown with itself</td>
<td>44</td>
</tr>
<tr>
<td>Proposition 7</td>
<td>Overlap and SMT solver</td>
<td>47</td>
</tr>
<tr>
<td>Proposition 8</td>
<td>Subsumption and SMT solver</td>
<td>48</td>
</tr>
<tr>
<td>Proposition 9</td>
<td>Tautology of order by conditions</td>
<td>58</td>
</tr>
<tr>
<td>Proposition 10</td>
<td>Loaded tuples of segment</td>
<td>59</td>
</tr>
<tr>
<td>Proposition 11</td>
<td>Not loaded tuples of segment</td>
<td>59</td>
</tr>
<tr>
<td>Corollary 12</td>
<td>Not loaded tuples of segments</td>
<td>60</td>
</tr>
<tr>
<td>Proposition 13</td>
<td>Score function based on combining function</td>
<td>64</td>
</tr>
<tr>
<td>Proposition 14</td>
<td>Order by condition of injective score function</td>
<td>64</td>
</tr>
<tr>
<td>Proposition 15</td>
<td>Order by condition with arbitrary score function</td>
<td>65</td>
</tr>
<tr>
<td>Proposition 16</td>
<td>Correctness of inclusion between histograms</td>
<td>110</td>
</tr>
<tr>
<td>Proposition 17</td>
<td>HS is NP-hard</td>
<td>113</td>
</tr>
<tr>
<td>Proposition 19</td>
<td>HS solvable by ILP</td>
<td>114</td>
</tr>
<tr>
<td>Corollary 20</td>
<td>HS is NP-complete</td>
<td>116</td>
</tr>
</tbody>
</table>