Data Science and Engineering (2025) 10:621–638
https://doi.org/10.1007/s41019-025-00290-1

RESEARCH PAPERS

# SOC: A Succinct Adaptive Semantic OLAP Caching

Jinguo You[1] · Yuxuan Wang[1] · Xingrui Huang[1] · Zhenrui Yi[1] · Wanting Fu[1] · Kaiqi Liu[1] · Pengchen Zhang[2] · Bin Yao[2]

Received: 22 April 2024 / Revised: 4 February 2025 / Accepted: 23 March 2025 / Published online: 10 June 2025


**Abstract**
In big data analysis, a large quantity of OLAP and aggregate queries exists, which have much stronger semantic context relationships (e.g. drill down and roll up) than generic SQL queries. Caching query results in memory playing an important role in accelerating data queries. Nevertheless, traditional query caching schemata neither fully utilize the features of OLAP, such as drill down and roll up semantics, nor compress the cached results, as the memory space is limited. In this paper, we propose a succinct, adaptive semantic OLAP caching, where the cache items are the cube lattice equivalence classes with only the bounds in a class stored. With further queries, the bound ranges are extended or expanded, indicating more query-answering ability which is assessed by the proposed covering capacity. The bounds of equivalence classes that more covering capacity are preferentially preserved in caching. We further empower our cache with some inference ability to derive more new data cells without posing extra queries and develop efficient query and update algorithms. The extensive experimental evaluation is conducted on synthetic and real data sets with various parameter settings. Our cache outperforms the common caching like LRU and LFU. Furthermore, it is robust to the non-repeated-pattern queries, still with a 30% hit ratio.

**Keywords** OLAP · Data caching · Query processing · Data cube lattice · Equivalence classes

## 1 Introduction

OLAP (On-Line Analytical Processing) is ubiquitous. It plays a fundamental role in big data analytics and data warehousing, having important applications in business management, marketing, and finance, etc. Real-world OLAP systems commonly employ caching techniques to accelerate OLAP query processing. It turns out that caching is crucial for high-performance OLAP query processing, especially in scenarios that share common queries and expect low latency [1, 2]. For example, Microsoft observes that over 45% of the daily OLAP jobs in their clusters have commonalities, resulting in millions of subexpression overlaps [3]. Caching such subexpressions can save up to 40% machine-hours. Alibaba Cloud also finds that a number of common computations exist in the analytical queries of their six projects. They then use cache to improve the performance of their OLAP database AnalyticDB [4].

✉ Jinguo You
jgyou@kust.edu.cn

[1] Kunming University of Science and Technology, 650000 Kunming, Yunnan, China

[2] Shanghai Jiao Tong University, 200030 Shanghai, China

Although there is quite a lot of related work devoted to query caching like page caching, tuple caching, and semantic caching [5], almost none of them considers compression by finding the associations of data items, specifically for aggregate queries of big data. OLAP and aggregate queries have much stronger semantic and context relationships than generic SQL queries, which are not yet fully exploited by most current caches.

Consider the base table sales in Table 1a, which has three dimensions: **Store** ($S$), **Products** ($P$), **Time** ($T$) and a measure **Sale**. We want to find the total sales in the context of different dimension combinations of $S$, $P$ and $T$. They are actually associated with 8 group-by views, which can be generalized by the data cube operator [1]. Each group-by view corresponds to a set of cells such as $(*, P2, *)$ and $(*, P2, T1)$, where $*$ means $ALL$. Each cell aggregates the tuples that satisfy the conditions over the group-by dimensions. Roll-up and drill-down are two basic semantic relationships of cells in a cube, with which a lattice structure is formed.

Now users posed a few queries like:
Select P, Sum(Sale) From sales Group by P;
Select P, T From sales Where P = P2 and T = T1;

**Table 1** The Base Table and Sample Query Items on It

(a) Base table

| No. | S | P | T | Sale |
|---|---|---|---|---|
| 1 | S1 | P1 | T1 | 6 |
| 2 | S1 | P2 | T1 | 12 |
| 3 | S2 | P1 | T2 | 9 |
| 4 | S3 | P3 | T2 | 3 |

(b) Query items

| No. | |
|---|---|
| 1 | (*,P2,*) |
| 2 | (*,P3,*) |
| 3 | (*,P2,T1) |
| 4 | (S1,P2,*) |
| 5 | (S3,P3,T2) |
| 6 | (S3,*,T2) |
| 7 | (S3,*,*) |
| 8 | (S3,P3,*) |
| 9 | (*,P3,T2) |

which actually correspond to the query items $(*, P2, *)$, $(*, P3, *)$, $(*, P2, T1)$ etc., shown in Table 1b. Traditional caches such as page caching, tuple caching, and semantic caching [5, 6] collect result items as tuples in fine granularity. That means they usually need to store all the result items, say 9 items in this case in Fig. 1a. Such high memory consumption makes it hard to scale to large-volume data under limited memory space. Furthermore, streaming data like web click streams with seldom repeated patterns makes the case worse, resulting in a bad hit ratio in caches.

We explore the semantics among OLAP query result items in our caching model. Some result items are identified as equivalence classes if they preserve drill-down and roll-up semantics and identical measure values, which we detail in Sect. 2 and Sect. 3. For each equivalence class, only the upper bound and lower bounds in the form of a lattice are retained in the cache. Any items between the upper and low bounds are also in the equivalence class and can be compressed.

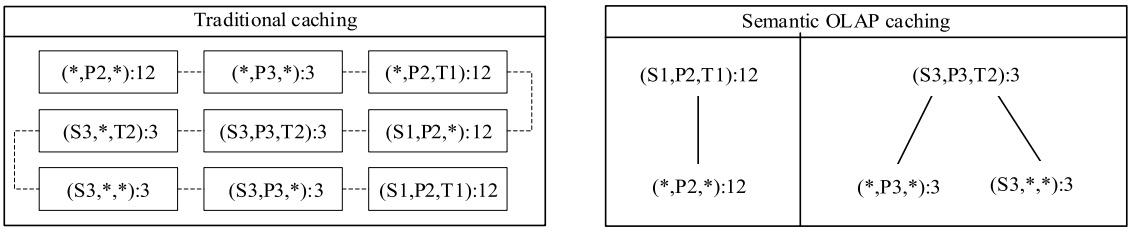
In Fig. 1b, our cache only stores 5 items, which form two equivalence classes, with a compression ratio of 44%. Most importantly, while achieving less memory consumption through compression, our method retains the same ability as traditional caching mechanisms to answer queries. The following example illustrates such a point.

**Example 1** *Assuming that the user poses a query Q*: *Select sum*(*Sale*) *From sales Where* $S = S3$ *and* $T = T2$.

The query in Example 1 corresponds to the query item $(S3, *, T2)$, which falls between the upper bound $(S3, P3, T2)$ and the lower bound $(*, P3, *)$ in the semantic OLAP cache. Therefore, the cache hits and the value of the measure 3 is returned. However, traditional caching fails to hit query items that it has never seen, say $(S3, *, T2)$. In addition, the semantic OLAP cache can also derive new data cells by using the known cache items. In Fig. 1b, the items $(S3, *, T2)$ and $(S3, P3, *)$ can infer the item $(S3, P3, T2)$ although it has never been seen before. Thus, more queries can be answered with cache, i.e., a higher hit ratio can be achieved.

Our main contributions are as follows:

- We propose novel OLAP caching techniques based on our theoretical analysis of equivalence classes. Instead of caching data tuples directly, our approach identifies similar tuples as an equivalence class and caches the lower and upper bounds of the classes. Our approach preserves the drill-down and roll-up semantics and could significantly reduce the memory overhead.
- We propose an adaptive caching mechanism by concluding six relations of the queries and the caching structures. They are query inclusion, upper bound extension, upper bound expansion, lower bound extension, lower




(a)Traditional caching model (b)Semantic OLAP caching model

**Fig. 1** Comparison between traditional caching and semantic OLAP caching

**Table 2** Common used notations

| Notation | Meaning |
|---|---|
| $\preceq$ | Roll-up |
| $\succeq$ | Drill-down |
| $\equiv$ | Equivalence |
| $\vee$ | The least upper bound |
| $\wedge$ | The greatest lower bound |
| $\|\|$ | Not connected/comparable |

bound expansion, and new class generation. The bounds are extendable or expandable with more queries, which implies the query-answering ability is enlarged with only a little space consuming.

- We present covering capacity in terms of the number of data cells in a class to assess the query-answering ability of the cache and then develop efficient query and update algorithms. A new cache replacement policy is proposed, where cache items with more covering capacity are preferentially preserved in the cache.
- We further empower our cache inference ability to some extent, i.e. to infer the data cells unseen before without submitting further queries. In essence, the conjunction of two data cells new to the cache is in the same class if these two cells are equivalent.
- The extensive experiments demonstrate the efficiency and effectiveness of our caching.

The paper is structured as follows. Section 2 reviews the main concepts and important properties of data cubes, particularly quotient cubes [7]. Section 3 presents our semantic OLAP cache model. We discuss the query match in the cache and the cache update and replacement in Sect. 4 and validate the semantic OLAP cache performance through an extensive experimental study in Sect. 5. The last section concludes the paper and describes future work.

## 2 Preliminary

In this section, we first describe the background of OLAP and then introduce some essential concepts in our approaches.

### 2.1 OLAP Background

Let $r$ be a relation over the database schema $R$. Attributes of $R$ are divided into two groups: (i) $D$ is the set of dimensions, and (ii) $M$ is the set of measures. $ALL$ (or $*$) represents a set in $D$, i.e., the set over which all the aggregate values were computed. A data cube $C$ over $r$ is calculated in the context of all dimension combinations. Note that in practice, data cubes selectively compute and store specific combinations of dimensions based on access frequency or computational cost, thereby reducing unnecessary computational overhead [8, 9]. Initially, our OLAP caching selectively preserves the queries with maximized covering capacity (detailed in Sect. 4.5) and adapts to query load instead of materializing the combination of all dimensions.

Let $c = (a_1, a_2, \ldots, a_{|D|}) : (m_1, m_2, \ldots, m_{|M|})$ be a data cell in a data cube, where $a_i$ is the value of a dimension $D_i \in D$ (include the value of $ALL$), $1 \leq i \leq \mid D \mid$, $m_j$ the value of a measure $M_j \in M, 1 \leq j \leq \mid M \mid$. The layer number of a data cell $c$, denoted by $\mid c \mid$, is the number of non-ALL dimension values in $c$. Obviously, we obtain $0 \leq \mid c \mid \leq \mid D \mid$.

Given two data cells $v, w \in C$, $v$ and $w$ have the following relationship, with notations and their meanings shown in Table 2:

**Definition 1** (partial order relation) If $\forall A \in D$, $v[A] \neq ALL \rightarrow v[A] = w[A]$, then $v$, $w$ have a partial order relation, denoted by $v \preceq w$ or $w \succeq v$. If $v \preceq w$ and $v \neq w$, then it is denoted as $v \prec w$ or $w \succ v$.

We can say that $v$ specializes $w$, or $w$ generalizes $v$. In other words, $v$ rolls up to $w$ or $w$ drills down to $v$. If $\forall A \in D$, $v[A] = w[A]$, then $v = w$. If $v$, $w$ do not satisfy $v \prec w$ or $v \succ w$, we say $v$, $w$ do not connect or comparable with each other, denoted by $v \mid\mid w$.

The partial order relation between data cells forms a cube lattice, where every pair of data cells $v$, $w$ has the least upper bound denoted by $v \vee w$ and the greatest lower bound of them denoted by $v \wedge w$.

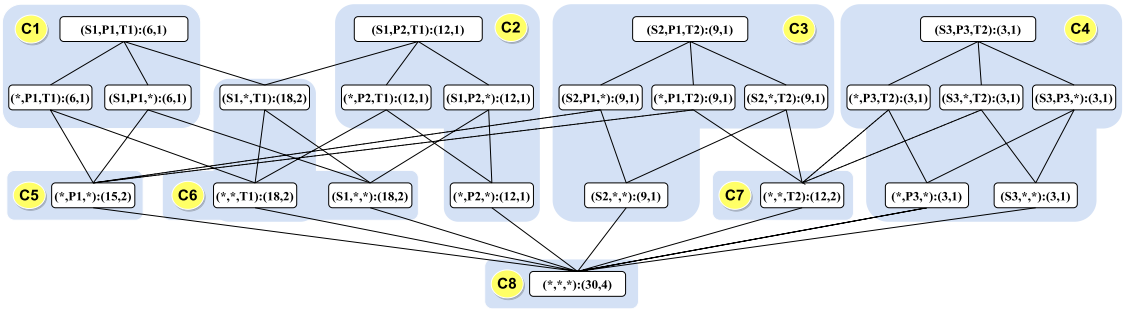
### 2.2 Essential Concepts

**Definition 2** (base tuple set) Given a data cell $v \in C$, the base tuple set of $v$, $BTS(v) = \{t \mid t \in r \text{ and } t \succeq v\}$, i.e. the set of all base table tuples that roll up to $v$.

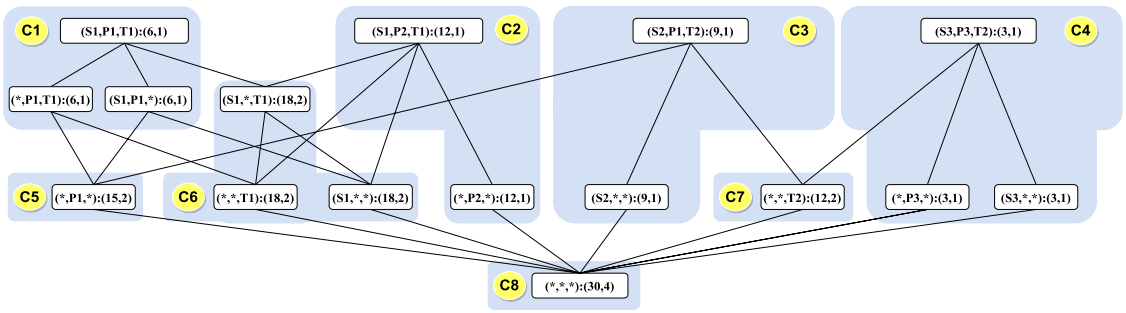
**Definition 3** (covering equivalence) The data cells $v$ and $w$ are equivalent, denoted by $v \equiv w$, if $BTS(v) = BTS(w)$.

Further, we have if $v \preceq w$, then $BTS(v) \subseteq BTS(w)$. Covering equivalence implies that for any aggregate function, the aggregate values of $v$ and $w$ are equal. But the converse does not always hold. Let's see the equivalence classes $C_2$ and $C_7$ in Fig. 2a. For the Sum aggregate function, the aggregate values of $(*, P2, *)$,$(*, *, T2)$ are all 12, but their base tuple sets are not equal.

**Lemma 1** (connecting equivalence) Given two data cells $v$ and $w$, if (i) the aggregate values at $v$ and $w$ are equivalent for a monotone function $f$, and (ii) there is a parent-child




(a) Quotient cube




(b) Simplified quotient cube

**Fig. 2** Quotient cube

relationship between $v$ and $w$, i.e. $v \preceq w$ or $w \preceq v$, then $v \equiv w$. Meanwhile, we say that the data cells $v$ and $w$ have the connecting equivalence relation.

***Proof*** Let $v \preceq w$, and we have $BTS(v) \subseteq BTS(w)$. Thus for a monotone function $f$, the aggregate values $m_v \leq m_w$, where $m_v = m_w$ holds when $BTS(v) = BTS(w)$. Since we have $m_v = m_w$, obviously we obtain $v \equiv w$.

Connecting equivalence is sufficient but not necessary for covering equivalence. That means data cells consistent with covering equivalence are not always connected. As shown in Fig. 2a, the data cells $(S1, P1, T1)$ and $(S1, P1, *)$ in the equivalence class $C1$ satisfies both connecting equivalence and covering equivalence between them. Because their base tuple set is the same, i.e., they all contain the same tuples in the base table that satisfy $S = S1$ and $P = P1$, they satisfy the coverage equivalence. Meanwhile, there is a parent-child relationship between them, i.e., $(S1, P1, *)$ is the parent node of $(S1, P1, T1)$. Along the lattice structure of the data cube, you can roll from $(S1, P1, T1)$ up to $(S1, P1, *)$, so the connection equivalence is also satisfied. In contrast, the data cells $(*, P1, T1)$ and $(S1, P1, *)$ in the equivalence class $C1$ satisfy the covering equivalence, but does not satisfies the connecting equivalence.

Note that we mainly consider the aggregate functions that are monotone in this paper since they include commonly used functions such as **Sum**,[1] **Max**, **Min** and **Count**, etc. We further discuss how to handle non-monotone aggregate functions in Sect. 4.6.

**Definition 4** (equivalence class) An equivalence class is a set of the data cells which are covering equivalent, denoted by $C = \{c \mid c_i \equiv c_j\}$.

A data cube can be partitioned into more than one equivalence class in terms of the covering equivalence relation.

**Lemma 2** The equivalence class $C$ is a convex set.

***Proof*** Given the data cells $u, l, v$, where $u, l \in C$ and $v$ satisfies $u \succeq v \succeq l$, we can infer that $BTS(u) \subseteq BTS(v) \subseteq BTS(l)$ in terms of $u \succeq v \succeq l$. Since $u, l \in C$, meaning $u \equiv l$, then $BTS(u) = BTS(l)$. Thus $BTS(v) = BTS(u) = BTS(l)$, and further $v \equiv u \equiv l$.

**Definition 5** (quotient cube) A quotient cube, denoted by $Q = (L/\equiv, \preceq)$, is a data cube lattice generated by

[1] when there are not both positive and negative values in the domain of measure attribute.

partitioning the data cells in the data cube in terms of the covering equivalence relation $\equiv$.

**Example 2** *Table 1b is a quotient cube constructed from the base table shown in Table 1a on the* **Sum** *aggregate function. Figure 2a shows a cube lattice which is partitioned in terms of equivalence classes. As we can see,* $(S1, P1, T1)$ *is the upper bound of the equivalence class* $C_1$*, while* $(*, P1, T1)$ *and* $(S1, P1, *)$ *are the lower bounds of the equivalence class* $C_1$*;* $(*, P1, *)$ *forms the equivalence class* $C_5$ *alone. To further simplify the view, Fig. 2b shows only the upper and lower bounds of the equivalent class in Fig. 2a.*

## 3 Our Approach

The formal definition of the semantic OLAP caching model is first given in this section. Then, we discuss how to store and manage a semantic OLAP cache.

### 3.1 Basic Idea

Optimized caching is to preserve a set of cache items with the most hit ratio under the caching space limitation. Here (i) we maintain the succinct semantic OLAP caching via finding the equivalent data cells in caching as an equivalence class that is convex, where only an upper bound cell and a few low bound cells are stored; (ii) we further iteratively preserve an equivalence class which contains the most equivalent data cells to answer the more queries while still keeping the succinct structure; (iii) the bounding of an equivalence class is dynamically extended or expanded with more query workload.

### 3.2 Semantic OLAP Cache Model

#### 3.2.1 Problem Formulation

**Definition 6** (cache item) A cache item $s$ is a subset of an equivalence class $C$, described by a triple tuple $s = \langle u, l, g \rangle$, where $u$ refers to the upper bound, $l$ refers to the lower bounds, and $g$ is the aggregate value list of $C$. Note that the number of the lower bounds may be more than one.

**Definition 7** (query item) A query item $q$ is a data cell posed by users with its aggregate values unknown.

**Definition 8** (semantic OLAP cache) A semantic OLAP cache is a set of cache items, defined as follows:

$$S = \left(s_1, s_2, \ldots, s_n\right)$$

We use $\mid S \mid$ to denote the size of the cache $S$, i.e. the number of all the data cells of upper bounds and lower bounds associated with the cache items.

Given a set of query items $Q = \left(q_1, q_2, \ldots, q_n\right)$, a semantic OLAP cache $S = \left(s_1, s_2, \ldots, s_n\right)$ where the size of each cache item $s_i$ is $w_i$. If a cache item hits a query item, then the obtained value of the cache item $v_i = 1$, if not, $v_i = 0$. Now we need to find a vector $x = \left(x_1, x_2, \ldots \ldots, x_n\right)$, $x_i \in \{0, 1\}$, $f(v, x) = \text{argmax} \sum v_i x_i$, s.t. $w_i x_i \leq \mid S \mid$.

#### 3.2.2 Relationship Between Cache Item and Query Item

To answer queries and form the semantic OLAP caching, we give the following relationship between cache item and query item: Given a cache item $s = \langle u, l, g \rangle$ and a query item $q$, we present their relationships as follows:

**Definition 9** (query inclusion) If $u \succeq q \succeq l$, then $q \equiv u \equiv l$. We say $s$ includes $q$ or $q$ is included by $s$.

**Definition 10** (upper extension) If $q \equiv u, q \succ u$, we say $q$ can extend $s$ since the upper bound of $s$ can be replaced by $q$ and the covering capacity of $s$ becomes greater.

**Definition 11** (upper expansion) If $q \equiv u$, $q \succ l, q \parallel u$, we say $q$ can expand $s$ since $q$ can be a new upper bound of $s$. Accordingly, the covering capacity of $s$ becomes greater.

**Definition 12** (lower extension) If $q \equiv u, q \prec l$, we say $q$ can extend $s$ since the lower bound of $s$ can be replaced by $q$ and the covering capacity of $s$ becomes greater.

**Definition 13** (lower expansion) If $q \equiv u$, $q \prec u, q \parallel l$, we say $q$ can expand $s$ since $q$ can be a new upper bound of $s$. Accordingly, the covering capacity of $s$ becomes greater.

**Definition 14** (a new class) If $q$ is unsatisfied with the other relationships, we say $q$ will generate a new class.

Figure 3 illustrates of the six relationships mentioned above from Fig. 3a–f. Figure 3f shows the case in which a new class is generated when none of the former five relationships is satisfied.

Note that the hit ratio depends on user query patterns, such as the repeat pattern or the non-repeat pattern. For the non-repeat pattern, traditional caching schemas are

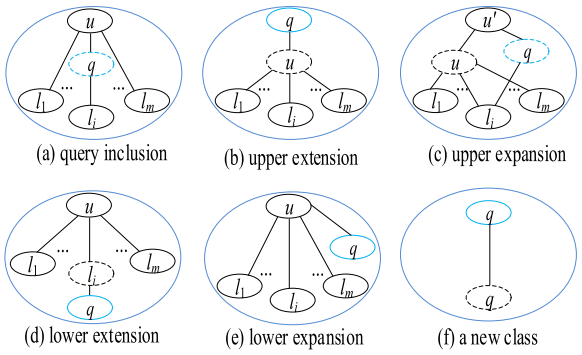



**Fig. 3** Relationship between cache item and query item

usually hard to hit, while our caching can match semantic equivalent query items via query inclusion. Query inclusion implies the hierarchical relationship among query items, which is essential in OLAP and caching, although the relationship is not always satisfied for some queries in the real world. Nevertheless, the experimental results in Sect. 5.3.2 shows the certain occurrence of the hierarchical relationship and, hence, the efficiency of query inclusion via hit ratio and query time for the non-repeat pattern.

The naive approach to query or update the cache is to scan every cache item to check whether it is equivalent to the query item. To improve the efficiency, we present the following theorem:

**Theorem 1** *The cache item $s = \langle u, l, g\rangle$ that is equivalent to the query item $q$ satisfies the nearest connection to $q$.*

***Proof*** We discuss two main cases: $u \succeq q$ and $u \prec q$ as follows:

$$
\begin{aligned}
1\,^{\circ} u \succeq q \Rightarrow & BTS(u) \subseteq BTS(q)\\
\text{if } \exists u', \mid u' \mid \geq \mid u \mid, u' \equiv q \Rightarrow & BTS(u') = BTS(q)\\
\Rightarrow & BTS(u) \subseteq BTS(u')\\
\Rightarrow & u \succeq u'\\
\Rightarrow & \mid u \mid \geq \mid u' \mid
\end{aligned}
$$

contradiction with $\mid u' \mid \geq \mid u \mid$.

$$
\begin{aligned}
2\,^{\circ} u \prec q \Rightarrow & BTS(u) \supseteq BTS(q)\\
\text{if } \exists u', \mid u' \mid < \mid u \mid, u' \equiv q \Rightarrow & BTS(u') = BTS(q)\\
\Rightarrow & BTS(u) \supseteq BTS(u')\\
\Rightarrow & u \preceq u'\\
\Rightarrow & \mid u \mid < \mid u' \mid
\end{aligned}
$$

contradiction with $\mid u' \mid \leq \mid u \mid$.

□

Theorem 1 indicates that if the cache item with the closest layer number is not equivalent to the query item, then we do not need to search the rest of the cache items.

### 3.3 Handling Unseen Queries

Besides caching historical queries, our approach can also be generalized to handle unseen queries.

By analyzing the data cells in a cache item, we find that some unseen data cells can be inferred from the known data cells in the cache under certain conditions.

**Theorem 2** *Given two data cells $l_1$, $l_2$, let their least upper bound $u = l_1 \vee l_2$, if $l_1 \equiv l_2$, then $u \equiv l_1 \equiv l_2$.*

***Proof***

$$
\begin{aligned}
u = l_1 \vee l_2 \Rightarrow & l_1 \preceq u, l_2 \preceq u\\
\Rightarrow & BTS(l_1) \cap BTS(l_2) = BTS(u)\\
l_1 \equiv l_2 \Rightarrow & BTS(l_1) = BTS(l_2)\\
\Rightarrow & BTS(l_1) = BTS(l_2) = BTS(u)\\
\Rightarrow & u \equiv l_1 \equiv l_2
\end{aligned}
$$

□

In other words, a new equivalent cell $u$ can be inferred without posing extra queries. Thus, more unseen query items are possibly answered, and the covering capacity is enhanced via either the above query inclusion or inferring a new cell.

**Corollary 1** There is one and only one upper bound in an equivalence class.

**Example 3** *As shown in Fig. 4, there are two data cells $l_1 = (*, *, 2)$ and $l_2 = (2, *, *)$ with the equivalence relationship in the cache. Then we infer that the least upper bound cell $u$ of $l_1$ and $l_2$ also belongs to the same equivalence class, although $u$ is not seen before.*

### 3.4 Covering Capacity

To maximize the semantic OLAP cache hit ratio, the query response ability needs to be evaluated. Intuitively, the more data cells it contains, the more likely it is to answer the queries. We use the covering capacity to assess the query response ability of our cache, which is as follows.

**Definition 15** (covering capacity) For a cache item $s$, its covering capacity, denoted by $ca$, is the capacity of covering the equivalent cells, i.e. the number of all the data cells in

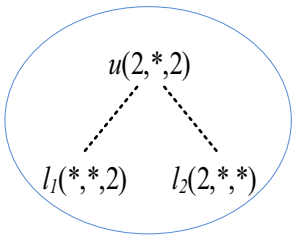



**Fig. 4** New data cell inference

between the upper bound and the lower bounds (including the upper bound and the lower bounds).

It can be known from Lemma 2 that the equivalence class is a convex set. That means all the data cells in between its upper bound and lower bounds are in the same equivalence class. In other words, if we know the upper bound and the corresponding lower bound in an equivalence class, We can infer all the data cells constrained by the bounds in terms of roll-up/drill-down semantics, which facilitates us to evaluate the covering capacity of an equivalence class.

When there is only one upper bound and one lower bound for an equivalence class $C$, we calculate its covering capacity as follows:

$$ca = \mid C \mid = 2^{|u|-|l|} \tag{1}$$

where $\mid u \mid$ is the layer number of the upper bound, and $\mid l \mid$ is the layer number of the lower bound.

As discussed above, there is always only one upper bound in an equivalence class, but there are usually multiple lower bounds.

Given an upper bound $u$ and lower bounds $l_1, \ldots, l_i, \ldots, l_n$ in an equivalence class $C$, $C_i$ is a set bounded by $u$ and $l_i$, denoted by $\{u : l_i\}$. We have $n$ sets respectively bounded by $\{u : l_1\}, \ldots, \{u : l_i\}, \ldots, \{u : l_n\}$. We evaluate the covering capacity of $C$ in terms of the inclusion–exclusion principle [10] as follows.

$$\begin{aligned} ca = |C| &= \left|\bigcup_{i=1}^{n} C_i\right| \\ &= \sum_{i=1}^{n} |C_i| - \sum_{1\le i\le j\le n} \left|C_i \cap C_j\right| + \ldots \\ &\quad + \sum_{1\le i\le j\le k\le n} \left|C_i \cap C_j \cap C_k\right| - \ldots \\ &\quad + (-1)^{n-1} \left|C_1 \cap \ldots \cap C_n\right| \end{aligned} \tag{2}$$

This can be compactly written as

$$\begin{aligned} ca = \mid C \mid &= \mid \bigcup_{i=1}^{n} C_i \mid \\ &= \sum_{\emptyset \ne J \subseteq \{1,2,\ldots,n\}} (-1)^{|J|-1} \mid \bigcap_{j\in J} C_j \mid \end{aligned} \tag{3}$$

Example 4 shows the calculation of the covering capacity of a cache item with multiple lower bounds.

**Example 4** *For $C_4$ in Fig. 2, there are 6 data cells, but in our cache item, the middle 3 data cells do not need to be saved. Thus there are only one upper bound $(S3, P3, T2)$, and two lower bounds $(*, P3, *)$ and $(S3, *, *)$ in the cache item. Assuming $C_1 = \{(S3, P3, T2) : (*, P3, *)\}$ and $C_2 = \{(S3, P3, T2) : (S3, *, *)\}$, the covering capacity can be calculated as follows*:

$$\begin{aligned} ca &= |C_1| + |C_2| - |C_1 \cap C_2| \\ &= 2^{3-1} + 2^{3-1} - 2^{3-2} = 6 \end{aligned}$$

*where* $C_1 \cap C_2 = \{(S3, P3, T2):(*, P3, *)\} \cap \{(S3, P3, T2):(S3, *, *)\}$ $\{(S3, P3, T2):(*, P3, *) \vee (S3, *, *)\} = \{(S3, P3, T2):(S3, P3, *)\}$.

## 4 Cache Management

### 4.1 Framework

In our semantic OLAP caching, when query items are matched in the cache, we index them according to the layer number. Figure 5 illustrates that if query items fail to hit, the cache system poses queries in the next level storage and then updates the cache with the new query results. Once the cache reaches its maximum capacity, the replace policy (Sect. 4.5) is applied to evict some cache items.

### 4.2 Cache Organization

Our cache items managed by the cache manager are equivalence classes. In addition, the cache manager is also responsible for query item matching, cache updating, and cache replacement in the cache.

The semantic OLAP cache is organized as shown in Table 3. Each cache item includes the main information: upper bound $u$, one or lower bounds $l$, aggregate values $g$, and the auxiliary information: covering capacity $ca$, number of data cell $sz$. In detail, $u$ and $l$ define a complete description of an equivalence class, while $g$ is used to store the query results. $ca$ represents the covering capacity of the equivalence class, that is, the number of data cells that the equivalence class can respond to. $sz$ represents the actual number of cells in the equivalence class.

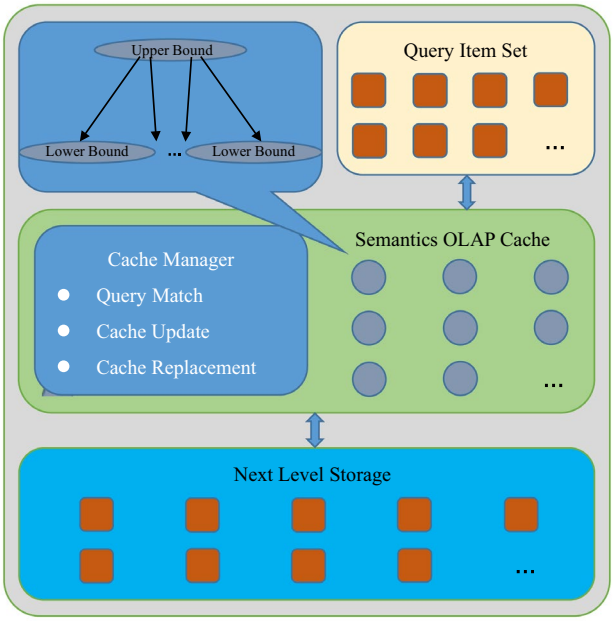



**Fig. 5** System architecture

**Table 3** The cached item structure

| Order | Items | Comments |
|---|---|---|
| 1 | $u$ | Upper bound |
| 2 | $l$ | One or more lower bounds |
| 3 | $g$ | Aggregate value |
| 4 | $ca$ | Covering capacity |
| 5 | $sz$ | Number of data cells |

## 4.3 Query Match

The query items submitted by users will first query the semantic OLAP cache. According to Lemma 2, a query item is hit when it is in between the upper and lower bounds of a cache item. Otherwise, it is not hit in the cache. For example, if $(S1, P2, *)$ is a query item, then the query is matched by the equivalence class bounded by $(S1, P2, T1)$ and $(*, P2, *)$ as shown in Fig. 1.

The query-matching algorithm is shown in Algorithm 1. In order to improve the matching efficiency, we first search the cache items according to the relationship between the layer number of the query items and the layer number of the cache items (line 3). For example, the two lower bounds $(*, P3, *)$ and $(S3, *, *)$ in $C_4$ in Fig. 2 have a layer number of 1, which is recorded in the $|\ l\ |_{min}$ of the cached item. In other words, the layer number of the cache items that can respond to the query item must be less than or equal to the layer number of the query item. Line 4–8 find the corresponding cached equivalence class if the query item is included between the bounds of the cached equivalence class.

**Algorithm 1** SOC_Query

**Input:** query items $Q$, semantic OLAP cache $S$
**Output:** query items' aggregate value
1 **Procedure**:
2 **for** *each* $q \in Q$ **do**
3 **for** $i = |\ q\ | \rightarrow |\ D\ |$ **do**
4 **for** *each* $s =< u, l, g > \in S_i$ **do**
5 $//S_i$ is the cache of the $ith$ layer
**if** $u \succeq q$ **then**
6 **if** $q \succeq l$ **then**
7 $q \equiv u$; //case a: query inclusion
return;
8 **else**
9 break; //no need to search remainder from current index $j$ to $|\ D\ |$

## 4.4 Cache Update

The storage granularity in the semantic OLAP cache is the equivalence classes. The cache is empty at the beginning of the query, and the equivalence classes need to be dynamically generated during the query. After the query item obtains the result in the next level of storage, it will generate or update the equivalence classes according to the connection equivalence relation.

**Algorithm 2** SOC_Update

**Input:** query item $q$ with its aggregate value $g'$
**Output:** updated semantic OLAP cache $S$

1 **Procedure**:
search by layer from the left side of q
**for** $i =\mid q \mid \rightarrow 0$ **do**
2 find $s =< u, l, g >\in S_j$ such that $g = g'$
**if** $l \prec q$ **then**
3 //connected by $u$
**if** $u \prec q$ **then**
4 $u = q$; //case b: extend upper bound
5 **else**
6 $u = u \vee q$; //case c: expand upper bound
7 **else**
8 break; //no need to search remainder from $\mid q \mid$ to $\mid D \mid$
9 //search by layer from the right side of q
**for** $i =\mid q \mid \rightarrow \mid D \mid$ **do**
10 find $s =< u, l, g >\in S_j$ such that $g = g'$
**if** $u \succ q$ **then**
11 //connected by $u$
**if** $\exists l_k \in l \succ q$ **then**
12 $l_k = q$; //case d: extend lower bounds
13 **else**
14 $l = l + \{q\}$; //case e: expand lower bounds
15 **else**
16 $S = S + \{q\}$; //case f: add a new class
return; //no need to search remainder from $\mid q \mid$ to 0

The cache update algorithm is shown in Algorithm 2. Given an answered query $q$ that is ready to be cached, we need to know which equivalence class it belongs for updating. Otherwise, it is created as a new cache item. In terms of Theorem 1, the possible class equivalent to $q$ is the one with the nearest connection to $q$. So we first search from the left side of $q$ (line 1) and then the right side of $q$ (line 9), as shown in Fig. 6. If an equivalent class is found on the left of $q$, we update, i.e. extend or expand, the corresponding upper bound that $q$ belongs to since all the cached upper bounds on the left with greater layers are more general than $q$. Specifically, if $q$ is covered by $l$, there are two cases: 1) $q$ is also covered by $u$, so we replace $u$ with $q$ as a new upper bound, extending the class; 2) else, we add $q$ as a new upper bound parallel to $u$, expanding the class. Accordingly, if an equivalent class is found on the right of $q$, we update, i.e. extend or expand, the corresponding lower bound.

Suppose in the initial case there is a cached equivalence class that contains an upper bound and one or multiple lower bounds, as shown in Fig. 3a. The query item $q$ has the following five cases when updating an equivalence class:

- When the query item $q$ is covered by the upper bound of the equivalence class, it replaces the original upper bound (line 1–4), and the equivalence class in Fig. 3a becomes the form of Fig. 3b.
- When the query item $q$ is covered by at least one lower bound of the equivalence class, the query item $q$ is added as the new upper bound in the equivalence class (line 5–6), and then the least upper bound $u' = q \vee u$ of $q$ and $u$ is inferred as the final unique new upper bound. The equivalence class in Fig. 3a becomes the form of Fig. 3c.
- When the query item $q$ can cover at least one lower bound in the equivalence class, it replaces the lower bounds that can be covered by the query item $q$ (line 9–12). If there is more than one lower bound that can be covered, the data cells will be compressed, and the equivalence class in Fig. 3a becomes the form of Fig. 3d.
- When the query item $q$ can cover the upper bound of the equivalence class but not any lower bound, the query item $q$ is added as the new lower bound in the equivalence class (line 13–14), and the equivalence class in Fig. 3a becomes the form of Fig. 3e.
- When the query item $q$ cannot update any of the equivalence classes, the query item $q$ is added to the cache as a new equivalence class (line 15–16), as shown in Fig. 3f.

Note that data cells in the same layer do not satisfy the covering equivalence relationship. In order to correctly generate the equivalence classes, some equivalence classes need to be merged. When the query item can serve as the upper bound or the lower bound of multiple equivalence classes, these equivalence classes need to be merged. Further, when there is more than one upper bound in the equivalence class, it triggers the inference so that there is always only one upper bound in the equivalence class.

In addition, to warm up the cache, we split the queries with the size 30,000 into two parts, with the first 20% to simulate

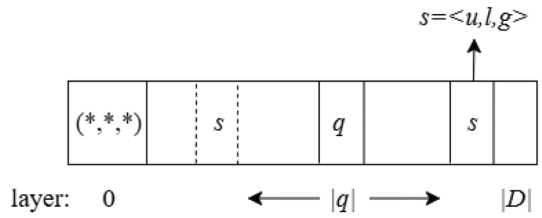



**Fig. 6** Cache update process

**Table 4** Comparison of preloaded query performance

| Performance | Not preload | Preload |
|---|---|---|
| Hit ratio | 0.6498 | 0.8155 |
| Caching query time(s) | 263 | 228 |

the user query patterns (that could be obtained by access frequency, access time or modeling in machine learning) to construct the initial equivalence classes, and the last 80% of the queries dynamically generate equivalence classes. As shown in Table 4, the preloading improved performance by 16% in hit ratio and 12% in caching query time.

## 4.5 Cache Replacement

As the memory space is limited, we need to preserve the most valuable items and evict the unnecessary items. Thus, the objective is to find $k$ data cells at most fitted in the cache that maximize the covering capacity, denoted by the following:

Let $k(k \in \mathbb{N})$ the maximum size of the cache, $S$ the semantic OLAP cache, and $ca(S)$ the sum of the covering capacity of all the cache items.

$$\begin{aligned} &\max \sum_{S} ca(S) \\ &\text{s.t.} \sum_{S} \mid S \mid \leq k \end{aligned} \tag{4}$$

It's a constrained optimization problem. For simplicity, we use the greedy algorithm to solve it. Specifically, the cache items with the maximum value of $ca$ are preferentially reserved. Besides, we hope to evict the class that cannot cover many queries but has lots of cells in the cache when the cache reaches its size limitation. So we introduce the average covering capacity (i.e., $\overline{ca}$). Let *size* denote the number of cells in a class, and we get:

$$\overline{ca} = \frac{ca}{size} \tag{5}$$

The $\overline{ca}$ can help evicting classes with the most inefficient $ca$.

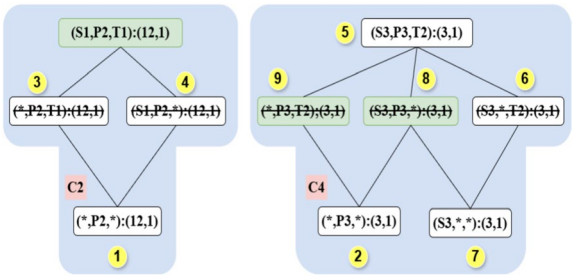



**Fig. 7** A running example

In Sect. 3.4, the way to calculate the cache item covering capacity is introduced. The effectiveness of covering capacity will be discussed in Sect. 5.3.5.

## 4.6 Non-monotone Functions

As discussed above, covering equivalence implies connecting equivalence, but connecting equivalence does not necessarily infer covering equivalence unless the aggregate function is monotone.

This paper mainly considers monotone aggregate functions, but it can be extended to more general cases, say non-monotone aggregate functions, in a simple way. First, we find the equivalence classes in terms of connecting equivalence on monotone aggregate functions, where covering equivalence are then guaranteed. Lakshmanan et al. [7] indicate that covering equivalence always produces a convex partition with the property that all cells in a class have the same aggregate value for any non-monotone aggregate function. If a query item with a non-monotone aggregate function falls into an equivalence class, it belongs to this class and can be answered (or update its non-monotone aggregate value if it does not exist).

## 4.7 A Running Example

Here we use some running examples to illustrate our idea above as follows. Assume that the query items arrive in the order of Table 1 (b) and the cache size is limited to 3.

### 4.7.1 Query Processing and Cache Updates

Figure 7 shows the walkthrough of a running example on Table 1, where the numbers indicate the order in which a user posts the queries.

1) First, the query items $(*, P2, *)$ and $(*, P3, *)$ are posed. Since initially the cache is empty, the queries are sent to the database and then results are returned and cached as the two new classes $C_2$ and $C_4$ respectively (case f: *new classes generation*).

2) Next, although the query item $(*, P2, T1)$ and $(S1, P2, *)$ are not hit in our cache, they can connect the the cache item $(*, P2, *)$ as the new upper bounds to expand the equivalence class $C_2$ (case c: *upper expansion*) once they are answered by the database. Further, they infer to a newer upper bound ($S1$, $P2$, $T1$).
3) The next query item ($S3$, $P3$, $T2$) also connects and expands $(*, P3, *)$ of $C_4$. Next, the class $C_4$ is expanded by the query item $(S3, *, T2)$ (case e: *lower expansion*) which is then extended by $(S3, *, *)$ as a newer lower bound (case d: *lower extension*).
4) According to Theorem 2, although the next posed query item $(S3, P3, *)$ is not observed before, it can be directly answered by the cached class $C_4$ as it is between ($S3$, $P3$, $T2$) and $(*, P3, *)$ of $C_4$ (case a: *query inclusion*). Finally, the next unseen query item ($S1$, $P2$, $T1$) is also hit in our cache.

#### 4.7.2 Range Query

A range query is a query that is performed on one or more dimensions of a data cube against a continuous range of data points. Suppose the user wants to perform a range query, say, for the total sales over the period from $T1$ to $T3$ on the time dimension T, with the query item $(*, P2, \{T1, T2, T3\})$. The equivalence class $C2$ has upper bound ($S1$, $P2$, $T1$) and lower bound $(*, P2, *)$. The query processing is as follows: First, check whether the equivalence class in the cache has an upper or lower bound containing this range query item. Then, we find that the query item $(*, P2, T1)$ is between the upper and lower bounds of the equivalence class $C2$, so the match is successful. Finally, We can fetch the total sales of 12 for $(*, P2, T1)$ directly from the cache without querying the database again.

#### 4.7.3 Iceberg Query

Iceberg queries materialize only those data cells that satisfy a specific minimum support threshold (*min_sup*). Based on the basic table above, suppose the user wants to perform an iceberg query to find the stores with sales of at least \$10 per product. The example below of an iceberg query posed on the cache is shown in Fig. 7.

```
Select S, P, T, Sum(Sale)
From base table
Group by P
Having Sum(Sale) ≥ 10;
```

This iceberg query essentially corresponds to three queries: $(*, P1, *)$, $(*, P2, *)$, $(*, P3, *)$ that satisfy Sum(Sale)$\geqslant$ 10. As Fig. 7 shows, $(*, P2, *)$ can be hit by our cache. Although $(*, P3, *)$ is possible be answered, its measure value is less than 10, which does not satisfy the condition of the iceberg query. As $(*, P1, *)$ cannot be hit, it needs to be submitted to the database for the query.

#### 4.7.4 Non-monotonic Aggregation

To deal with non-monotonic aggregation query, particularly for Avg aggregation function, our cache stores both Sum and Count values that are monotonic. Based on the above cache, assume a user poses query item $(*, P3, T2)$ but on **Avg** aggregate function that is non-monotonic. As data cell $(*, P3, T2)$ is included by $C4$, it can be still hit by our cache if we know its **Sum** and **Count** aggregate value of the equivalence class $C4$. The specific steps are: 1) Get the Sum and Count values of $(*, P2, *)$ from the cache; 2) Calculate the Average value: Average = Sum / Count; 3) Return the calculation result: Average = 3 / 1 = 1.

## 5 Performance Evaluation

### 5.1 Experimental Settings

To evaluate the performance of the semantic OLAP cache, we conducted a comprehensive experiment, our experimental platform is as follows.

#### 5.1.1 Environment Setup

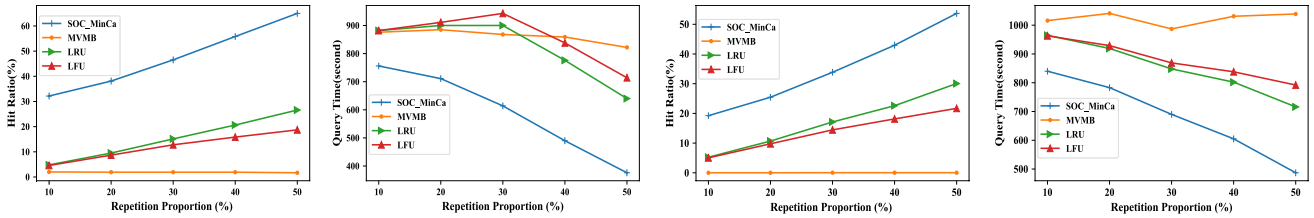
The experiments are running on a laptop with 2.2 GHz Intel Core i7, 256 GB SSD and 16 GB RAM. The database is SQLite since it's a light and fast SQL engine. Note that MySQL or other SQL engines are alternative experimental platforms. All the programs are coded using C++ in XCode 10.1.

#### 5.1.2 Datasets

Two different data sets are used: 1) the weather, which is a standard synthetic data set frequently calibrating various cube algorithms [7, 11]. 2) the lineitem table in TPC-H, which is a real data set widely used in OLAP benchmarks. For the weather data set, we selected 10 dimensions: *time*, *LAT*, *ID*, *ww*, *AM*, *AH*, *SA*, *RI*, *SLP*,*WS* and a measure *LON* with 500,000 tuples. For the synthetic data set, we selected five dimensions: *l_orderkey*, *l_partkey*, *l_suppkey*, *l_linenumber*, *l_quantity*, and a measure *l_extendedprice* from the *lineitem* table with 1,000,000 tuples.

#### 5.1.3 Query Workloads

To simulate the user's query workload, we set 20% of the tuples in the base table as a hot region and 80% as a cold region. 90% of the query data are randomly generated from




(a) TPC-H hit ratio (b) TPC-H query time (c) Weather hit ratio (d) Weather query time

**Fig. 8** Hit ratio and query time on TPC-H data (tuple size: 1,000,000, cache size: 8,000) and weather data (tuple size: 500,000, cache size: 9000) w.r.t. repeated-pattern proportion

the hot region tuples, and 10% are modified from the cold region as configured in [5, 6]. In our experiments, we mainly used the point query since it is the basic query from which other queries, like the range query or the iceberg query, can be derived.

#### 5.1.4 Baselines

We compared our semantic OLAP cache (SOC_MinCa) mainly with the traditional LRU and LFU, materialized view selection by greedy algorithm (MVMB)[8], which are all fully implemented. Note that when using LRU, LFU, and MVMB algorithms, the entities managed by the cache manager are data cells or data tuples rather than equivalence classes since they do not consider the semantic relationship among data tuples. The semantic cache presented in these two references[5, 6] does the same except for adding a constraint formula represented by the SQL predication description of the tuples. They are all simulated by the tuple caching.

### 5.2 Performance Metrics

To evaluate the performance of SOC, we use four different metrics: query time, hit ratio, compression ratio, and covering capacity.

Query time measures the time it takes for the database to respond after the query request is made.

While query time is an important metric for the OLAP system, hit ratio is also a significant metric for cache, which means during the execution of a query, let $c$ represent the number of cache hits, $m$ represent the number of total queries, and $h$ represent the cache hit ratio, it is defined as follows:

$$h = \frac{c}{m} \tag{6}$$

Here we define the compression ratio as $C$ means the difference between the traditional query cache size and the semantic OLAP query cache size, and then divided by the traditional query cache size, let $t$ represent the traditional query cache size and $s$ represent the semantic OLAP query cache size, it is defined as follows:

$$c = \frac{t - s}{t} \tag{7}$$

Covering capacity is also an important metric. We give a more detailed definition in section 4.4.

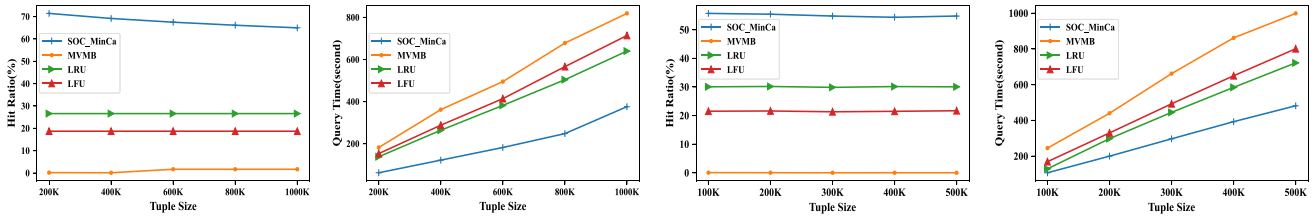
### 5.3 Experimental Results

#### 5.3.1 Repeated-Pattern Query

This experiment verified the case of repeated-pattern queries, which means that the query item has some duplication. Some duplicate data was added in the query items in different proportions to make some query patterns recur.

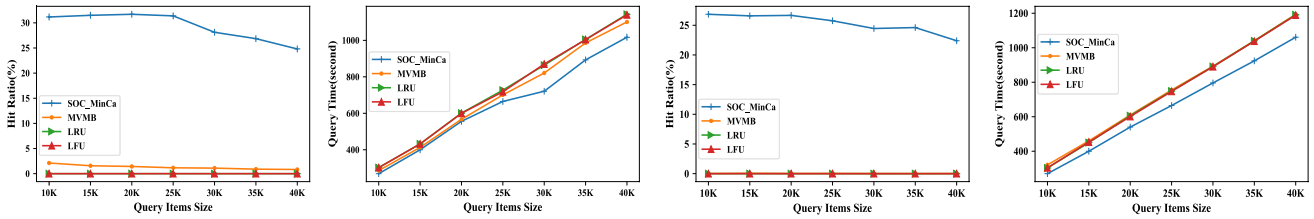
First, we validated the performance when the proportion of repeated queries varied. The initial size of the query items is 30,000, and the duplicate data is derived from them in the proportion of 10, 20, 30, 40, and 50 with the 2 duplicate copies of each query item. Figures 8a and c show that the hit ratio of the SOC_MinCa algorithm is higher or closer to the repetition proportion of the query items, In contrast, the hit ratio of the LRU, LFU and MVMB algorithms is lower than that of the query items. The reason is that SOC_MinCa can answer not only the seen historical query items that are repeated, but also the unseen query items (see the section 3.3). However, the LRU and LFU simply store the data cells in the cache without exploring the semantics among the data cells. The MVMB algorithm, as it is a static algorithm, does not consider the dynamic query workload and the semantics among query items. Thus, the hit ratio of MVMB is at a very low level, around 1%. Figure 8b and d show the query time on the repeated-pattern query. It can be seen that SOC_MinCa is about 15% faster than the baselines.

Second, we validated the performance when the number of the data tuples varied. Figures 9a and c show that as the number of data tuples increases, the SOC_MinCa algorithm's hit ratio can still be higher than the LRU and LFU




(a) TPC-H hit ratio (b) TPC-H query time (c) Weather hit ratio (d) Weather query time

**Fig. 9** Hit ratio and query time on TPC-H data (cache size: 8000) and weather data (cache size: 9000) w.r.t. tuple size (repeated-pattern proportion: 50%, query data size: 30,000)




(a) TPC-H hit ratio (b) TPC-H query time (c) Weather hit ratio (d) Weather query time

**Fig. 10** Hit ratio and query time on TPC-H data (tuple size: 1,000,000, cache size: 8,000) and weather data (tuple size: 500,000, cache size: 9000) w.r.t. query item size (repeated-pattern proportion: 0%)

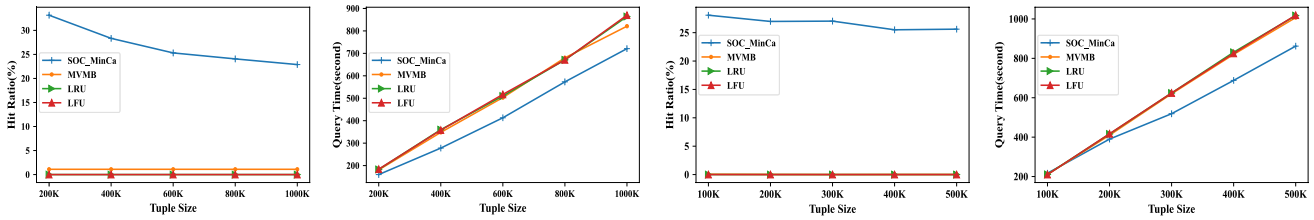
algorithms. However it declined in the later period because the number of tuples increases. Figures 9b and d show that the query time of the SOC_MinCa algorithm is consistently shorter than the LRU, LFU, and MVMB algorithms, with about 5% to 50% relative improvement.

### 5.3.2 Non-repeated-pattern Query

We also verified the case of non-repeated-pattern queries. In big data scenarios, their size far exceeds the size of the cache, so retaining the query patterns is most difficult or impossible. However, our semantic OLAP cache can still answer the queries because our cache fully explores the semantics among query items. This is the reason why our semantic OLAP cache can be applied in big data scenarios.

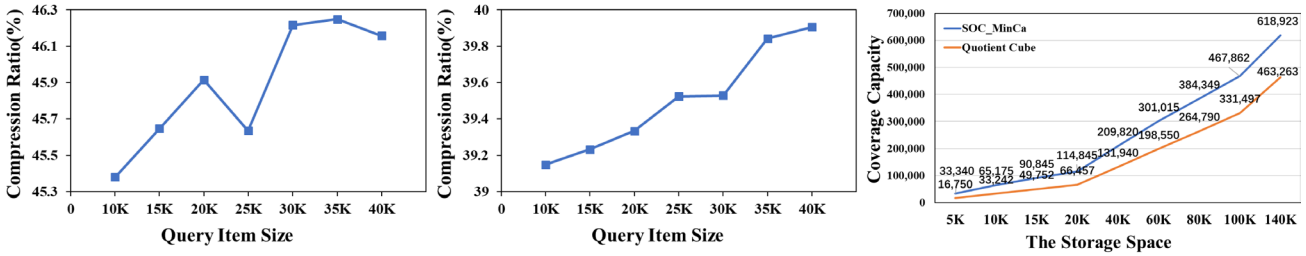
First, we validated the performance when the number of query items varied. In the experimental setting, the query items fed in the fixed-size cache are not repeated, and the sizes varied from 10,000 to 40,000. Our cache completely outperforms the traditional cache. The hit ratio with the non-repeated-pattern query is shown in Fig. 10a and c. As the number of query items increases, the more complete cache items are established, because our cache items can preserve semantics. So in this case of non-repeated-pattern queries, our caching reaches a rather reasonable hit ratio of around 20–30%, but for LRU and LFU algorithms that cannot preserve semantics, their hit ratio is always 0 (the results of LRU and LFU overlap in the figure). Although the MVMB algorithms, they can successfully hit a few query items, the hit ratio of the materialized view is 0.1%, much lower than our SOC_MinCa algorithm in the limited cache. Note that in the SOC_MinCa algorithm, the hit ratio is dropped in the later period because the cache limit has worked. After all, the cache limits of 9,000 and 8,000 are too small for 40,000 query items. To address this issue, our caching system is equipped with a dynamic cache expansion mechanism such as doubling the current cache size if available that allows it to increase capacity according to the query load adaptively. This ensures that even with a large number of query items, the system will have a high hit rate and manage the caching process efficiently. A higher hit ratio means that caches can save time when looking up results. The results in Fig. 10b and d show that the running time of the SOC_MinCa algorithm is better than that of the LRU and LFU algorithms. In summary, the SOC_MinCa algorithm not only has a high hit ratio compared with the hit ratio 0 of the LRU and LFU but also has a better running time.

Second, we validated the performance when the number of the data tuples gradually varied. Figure 11a and c show that the SOC_MinCa can still answer the queries even if no repeated-pattern data are input, while the LRU and LFU algorithms cannot hit at this time. The results




(a) TPC-H hit ratio (b) TPC-H query time (c) Weather hit ratio (d) Weather query time

**Fig. 11** Hit ratio and query time on TPC-H data (query item size: 30,000, cache size: 8000) and weather data (query item size: 30,000, cache size: 9000) w.r.t. tuple size (repeated-pattern proportion: 0%)




(a) TPC-H compression ratio (b) Weather compression ratio (c) TPC-H covering capacity

**Fig. 12** Compression ratio on TPC-H data and weather data and covering capacity comparison with quotient cubes

of Fig. 11b and d show that the SOC_MinCa outperforms the LRU, LFU, and MVMB algorithms in the query time.

### 5.3.3 Cache Compression

In this section, we use the unlimited size cache and the non-repeated-pattern query items to evaluate the compression ratio and the covering capacity of different caches. The data items are not replaced because the cache is unlimited but with the cache dynamic update described in Sect. 4.3, our cache will automatically delete some redundant data cells when updating, and such deletion will not weaken the semantics of the cache item.

The compression ratio is shown in Fig. 12a and b. In the experiment, both the LRU and LFU algorithms do not compress the cache, so the compression ratio is 0 (the results of LRU and LFU overlap in the figure). The SOC_MinCa algorithm can replace data cells during the update of the cache, so the SOC_MinCa algorithm can reduce data even without limiting the cache size.

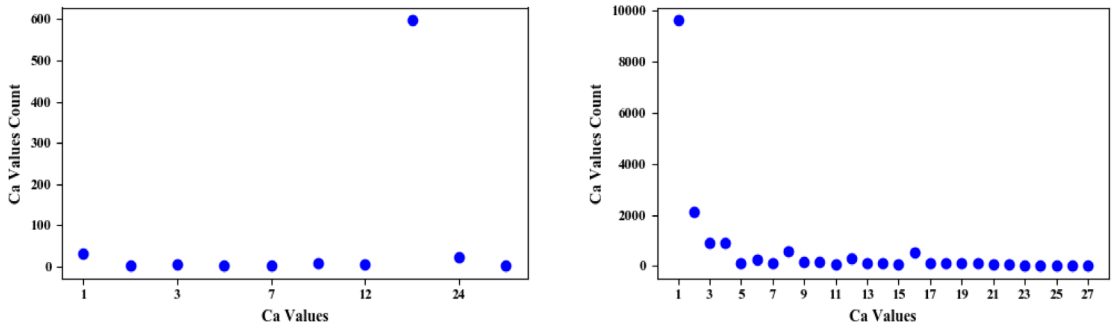
In addition, we compared with quotient cubes on the TPC-H benchmark data set for the compression of data cubes. Part of the results are shown in Fig. 12c. When the storage space size is fixed, they all compress some data cells and thus answer more query items than what the fixed storage contains. However, compared with the quotient cube compression method, our caching always maintains the higher covering capacity and then the higher query ability. The reason is that we preferentially keep the equivalence classes that have maximal covering capacity, instead of the trivial equivalence classes.

### 5.3.4 Improved LRU and LFU with Equivalence Classes

Since the traditional LRU and LFU algorithms have a compression rate of 0, we incorporate the idea of equivalence class compression from the SOC_MinCa algorithm with into

**Table 5** Preloading Performance Comparison of Caching Algorithms

| Performance | | SOC_MinCa | SOC_LRU | SOC_LFU |
|---|---|---|---|---|
| Not preload | Hit ratio | 0.6498 | 0.266 | 0.1871 |
| | caching query time(s) | 263 | 621 | 685 |
| Preload | hit ratio | 0.8155 | 0.4919 | 0.4923 |
| | caching query time(s) | 228 | 519 | 578 |




(a) dataset1 skew 0.5 (b) dataset2 skew 2

**Fig. 13** *ca* distributions on various data set

LRU and LFU algorithms. The improved LRU and LFU algorithms also achieve the corresponding cache compression capability. The difference among them is thus mainly in the cache replacement strategy. Our cache replacement strategy is to keep the equivalence classes with the largest coverage capacity and evict the equivalence classes with small coverage capacity, especially the possible singleton equivalence classes. In contrast, the cache strategy of the LRU algorithm is to eliminate the data items that have not been accessed for the longest time, and that of the LFU algorithm is to eliminate the data items with the fewest accesses. In the same experimental environment, we compare the improved LRU and LFU algorithms with the proposed SOC_MinCa algorithm. Table 5 shows the query performance comparison between different caching algorithms with and without preloading. The SOC algorithm shows the highest hit rate and the shortest cache query time in both cases, which indicates that it is more efficient in cache management. This is because maintaining the maximum coverage capacity in SOC_MinCa tends to answer more queries with higher possibility.

### 5.3.5 Effect of Covering Capacity

The method of calculating the cache item covering capacity *ca* using the set inclusion–exclusion principle is introduced in Sect. 3.4. Covering capacity (*ca*) can intuitively and accurately reflect the responsiveness of cache items. Since the cache item can preserve semantics, the covering capacity (*ca*) refers to the number of data cells that the cache item can answer instead of the number of actually saved data cells (*sz*). For example, in the equivalence class $C_6$ of Fig. 2, *ca* and *sz* are both 1, and in the equivalence class $C_4$, *ca* is 6 and *sz* is 3. In addition, when the *ca* value of the equivalence class is 1, the equivalence class cannot preserve semantics, which, in essence, directly places the data cell in the cache. As can be seen from Sect. 3.4, the more the lower bounds are in the cache item, the more items are calculated. However, we observed that the number of the lower bounds is small in practice, so the overhead of calculating *ca* is acceptable by using the principle of inclusion–exclusion.

We know that a quotient cube is derived from the base table via aggregates. In other words, the form of the equivalence classes and the covering capacity of a quotient cube are closely related to the data distribution of the base table.

**Table 6** Dataset configuration

| Items | dataset1 | dataset2 |
|---|---|---|
| Dimension number | 5 | 5 |
| Base table size | 5000 | 5000 |
| Dimension skew | 0.5 | 2 |
| Dimension 1 cardinality | 160 | 357 |
| Dimension 2 cardinality | 121 | 340 |
| Dimension 3 cardinality | 136 | 361 |
| Dimension 4 cardinality | 129 | 333 |
| Dimension 5 cardinality | 134 | 327 |

We designed a set of experiments to illustrate it. Two synthetic datasets were used in the experiment. The maximum value of each dimension is 500,000. The configuration is detailed in Table 6. To observe the distribution of *ca* values after querying on two datasets, it is necessary to generate equivalence classes as fully as possible when querying. The dataset1 generates 10,533 query items and the dataset2 generates 6,8031 query items. We found that the skewer the dimension values in the base tables are, the more the equivalence classes with the less value of *ca*, especially the singleton equivalence classes, are generated (Fig. 13). This is because the duplicate dimensional values become more as the skewness increases, which will generate the equivalence classes involving multiple base tuples and thus decrease the

*ca*. Compared with the quotient cube compression method, our method preferentially retains the equivalent class with the largest ca value, that is, the ones with the smaller ca especially including the singleton one are evicted, so our cache always maintains a high coverage capacity, and thus maintains a high query capacity.

## 6 Related Work

According to the type of cached objects, traditional caching techniques can be divided into three kinds: page caching, tuple caching, and semantic caching. Page caching ships and stores the data in page-size units, while tuple caching stores the individual tuples at a finer granularity. In contrast, semantic caching stores not only the result data but also the data semantic description mainly represented by SQL predictions. Note that the semantics in our semantic OLAP cache refers to the (aggregate) data relationship like drilling-down or rolling-up, not the data description.

These caching kinds have attracted a lot of interest from the database research communities. Page caching and tuple caching were compared in [5]. Dual Buffering [12] and Hybrid Caching [13] are proposed respectively to exploit the advantages of both page caching and tuple caching. Semantic caching was proposed in [5]. [14] gave a logic semantic caching model based on Datalog and the query matching criteria. The more formally defined semantic caching model was proposed in [6], which focuses on Select-Project-Join queries (short for SPJ). The recent works are semantic caching over cloud [15, 16].

Data cube operator [17] was first proposed by Jim Gray in 1997. Due to the space overhead of data cubes or materialized views, considerable work has been devoted to their efficient computation. Materialized view selection (MVS) is a classic problem, where only part of the data cube or views are materialized [18–20]. Along with the view dependency lattice, [21] proposed greedy partial materialization algorithms in terms of the benefits gain if $k$ views are materialized in contrast to materializing from the top view. The benefit is evaluated mainly based on the linear cost model, namely the number of tuples in the base table for that query. [9] further proposed the more efficient algorithm PBS (Pick By Size). Like the caching, [22] proposed DynaMat, which dynamically selects and maintains optimal materialized views according to the incoming queries. Recently, [23] proposes an autonomous materialized view method by deep reinforcement learning.

On the other hand, there have been some efforts aiming at compression or summarization of data cubes. They can fall into three main categories: 1) lossless compression or summarization: BUC cubing [24], Condensed cube [25], Dwarf cube [26], Quotient cube [7] and QC-Trees [11], etc.; 2) lossy compression: Quasi cube [27], Wavelet Cube [28]; 3) hybrid: the most recent work [29] explores the lossless and lossy summarization of hierarchical multidimensional data.

The main current works are: 1) extending cubing algorithms to distributed computing environments by applying the Hadoop or Spark framework etc. [30–33] and 2) OLAPing on non-structured data or streaming data, say text cube [34], information networks [35], PopUp-Cubing [36].

Most of the above methods are static and seldom adaptable to the query workload. In view of the importance of OLAP, especially big data query and analysis which involves lots of join and aggregate operators, the efficient algorithms and applications of OLAP itself still need to be studied thoroughly.

We note the interesting links between materialized view selection and data cube compression based on the earlier work of containment and equivalence of conjunctive SQL queries [37, 38] which were further developed by [39, 40] etc. In essence, they are all queries requested by users and correspond to a set of tuples in the base table. [18, 37] mentioned that given two queries $Q_1$, $Q_2$, they are equivalent if the set of tuples in the database computed for them are identical, and further $Q_1$ is contained by $Q_2$ if the set of tuples computed for $Q_1$ is a subset of those computed for $Q_2$. We found that the dependencies of queries along with the lattice framework can be built in terms of containment. Moreover, the equivalence of queries is actually the covering equivalence in [7, 11]. Nevertheless, the containment and equivalence in materialized views are mainly exploited in query rewrite and optimization, not in compression. In fact, many group-by views may be equivalent, say in Fig. 2.

The most related literature is [7, 11], which extracts the semantics from a data cube to obtain a concise summary of it. Continuing this semantic meaning, this paper explores it in aggregate query cache scenarios. But they have different flavors in some aspects which are elaborated as follows.

1) The quotient cube is constructed from the base table. In contrast, our caching is driven by user queries, i.e. it finds equivalence classes by summarizing the semantics on the fly once users pose queries. Users need not practically construct a whole quotient cube from the base table before caching. This makes it harder to judge the equivalence classes than the quotient cube. The corresponding query and update are nontrivial and need more tricks. We propose six relations between the queries and the caching, including query inclusion, upper bound extension, upper bound expansion, lower bound extension, lower bound expansion, and a new class generation (in Sect. 3.2). In addition, the efficient cache query and update algorithms are developed.

2) Our OLAP caching is more succinct than quotient cubes under the constraint of limited storage spaces. Given $n$ equivalence classes, we only select $k$ equivalence classes with maximal covering capacity while still preserving high query answering ability or hit ratio. We demonstrate our caching attains higher overall covering capacity and query answering ability than quotient cubes in Sect. 5.3.3.

## 7 Conclusion and Future Work

In this paper, we propose a novel and promising caching aiming at OLAP and aggregate queries.

Compared with other caches that do not consider the possible space redundancy, our cache model keeps a succinct cache structure that only stores an upper-bound cell and the lower bound cells of a set of equivalent cells. With more queries posed by users, the cache structure is dynamically adapted and the covering capacity is expanded, implying the ability to answer more queries without increasing the extra memory space. The efficiency of our semantic OLAP caching is demonstrated by the extensive experiments. Note that our focus in this paper is to provide and validate the features of the compact and adaptability of OLAP query cache.

Next, we will optimize our caching algorithms, such as cache update (essentially equivalence classes update), to keep consistency with base table updates and extend it to incremental maintenance of data cubes with bulk changes in the underlying tables. The semantics we have preserved provide a way to update the cache model according to the query items that are derived from the updated tuples. If an inserted or deleted tuple has the same dimension values as an existing tuple, classes whose cells cover the tuple should recompute their aggregation values to reflect the data update. If there is no tuple in the table that has the same dimension values as the newly updated tuple, the classes should be split or merged depending on whether the tuple manipulation is an insertion or deletion. If the updating of a tuple only modies an existing tuple, the classes simply need to recompute their aggregation values.

Note that at present our SOC caching model supports select, project, aggregate, and group by, etc., relational operators on single tables. So for another future research direction, we plan to extend the succinct cache model to include more complicated queries, such as joins that involve multiple tables. One simple solution for join queries is to regard the result of the join operation as a single table and generate equivalence classes for it.

In addition, our SOC caching model is intended to couple with the caching or buffering of existing databases such as PostgreSQL, MySQL, and openGauss for generic SQL queries.

**Acknowledgements** The authors wish to thank the anonymous reviewers for their helpful feedback. The authors are grateful to Jianqing Xi for his support at the South China University of Technology, Guangzhou, China. This work was partially supported by the Natural Science Foundation of China (62062046, 62262035).

**Author Contributions** Jinguo You makes contributions on the ideas and paper writing. Yuxuan Wang, Xingrui Huang and Zhenrui Yi make contributions on implementation of models and experiments. Wanting Fu and Kaiqi Liu make contributions on paper writing and revising, experiments conducting. Pengchen Zhang and Bin Yao also make contributions on paper writing.

**Data availability** https://github.com/bizard-lab/soc_cache.



## References

1. Durner D, Chandramouli B, Li Y (2021) Crystal: a unified cache storage system for analytical databases. Proc VLDB Endow 14(11):2432–2444
2. Zarif OE, Hassan S, Zou Y, Zuzarte C, Corvinelli V, Alhamid M (2020) Pred-cache: a predictive caching method in database systems, pp. 93–102
3. Jindal A, Karanasos K, Rao S, Patel H (2018) Selecting subexpressions to materialize at datacenter scale. Proc VLDB Endow 11(7):800–812
4. Zhan C, Su M, Wei C, Peng X, Lin L, Wang S, Chen Z, Li F, Pan Y, Zheng F, Chai C (2019) AnalyticDB: real-time OLAP database system at Alibaba cloud. Proc VLDB Endow 12(12):2059–2070
5. Dar S, Franklin MJ, Jónsson B, Srivastava D, Tan M (1996) Semantic data caching and replacement. In: 1996 International Conference on Very Large Data Bases, pp. 330–341
6. Ren Q, Dunham MH, Kumar V (2003) Semantic caching and query processing. IEEE Trans Knowl Data Eng 15(1):192–210
7. Lakshmanan LV, Pei J, Han J (2002) Quotient cube: How to summarize the semantics of a data cube. In: 2002 International Conference on Very Large Data Bases, pp. 778–789
8. Kumar T, Haider M, Kumar S (2011) A view recommendation greedy algorithm for materialized views selection. Commun Comput Inf Sci 141:61–70
9. Shukla A, Deshpande P, Naughton JF (1998) Materialized view selection for multidimensional datasets. In: 1998 International Conference on Very Large Data Bases, pp. 488–499
10. Wikipedia contributors: Inclusion-exclusion principle — Wikipedia, The Free Encyclopedia. (2019). https://en.wikipedia.org/wiki/Inclusionexclusion_principle

11. Lakshmanan LV, Pei J, Zhao Y (2003) QC-trees: an efficient summary structure for semantic OLAP. In: Proceedings of the 2003 ACM SIGMOD, pp. 64–75
12. Kemper A, Kossmann D (1994) Dual-buffering strategies in object bases. In: 1994 International Conference on Very Large Data Bases, pp. 427–438
13. Wu X, Li J, Zhang L, Speight E, Rajamony R, Xie Y (2009) Hybrid cache architecture with disparate memory technologies. In: International Symposium on Computer Architecture, pp. 34–45
14. Godfrey P, Gryz J (1999) Answering queries by semantic caches. In: International Conference on Database and Expert Systems Applications, pp. 485–498
15. Ahmad M, Qadir MA, Rahman A, Zagrouba R, Alhaidari FA, Ali T, Zahid F (2023) Enhanced query processing over semantic cache for cloud based relational databases. J Ambient Intell Humaniz Comput 14(5):5853–5871
16. Ma K, Bo Y, Zhe Y, Yu Z (2017) Segment access-aware dynamic semantic cache in cloud computing environment. J Parallel Distrib Comput 110:42–51
17. Gray J, Chaudhuri S, Bosworth A, Layman A, Reichart D, Venkatrao M, Pellow F, Pirahesh H (1997) Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotals. Data Min Knowl Disc 1(1):29–53
18. Halevy AY (2001) Answering queries using views: a survey. VLDB J 10(4):270–294
19. Srinivasarao P, Satish AR (2023) A novel hybrid optimization algorithm for materialized view selection from data warehouse environments. Comput Syst Sci Eng 47(2):1527–1547
20. Mohseni M, Sohrabi MK (2020) MVPP-based materialized view selection in data warehouses using simulated annealing. Int J Coop Inf Syst 29(3):2050001–1205000119
21. Harinarayan V, Rajaraman A, Ullman JD (1996) Implementing data cubes efficiently. In: Proceedings of the 1996 ACM SIGMOD, pp. 205–216
22. Kotidis Y, Roussopoulos N (1999) DynaMat: a dynamic view management system for data warehouses. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, pp. 371–382
23. Han Y, Li G, Yuan H, Sun J (2021) An autonomous materialized view management system with deep reinforcement learning. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE)
24. Beyer K, Ramakrishnan R (1999) Bottom-up computation of sparse and iceberg cube. In: Proceedings of the 1999 ACM SIGMOD, pp. 359–370
25. Wang W, Feng J, Lu H, Yu JX (2002) Condensed cube: an effective approach to reducing data cube size. In: Proceedings 18th International Conference on Data Engineering, pp. 155–165
26. Sismanis Y, Deligiannakis A, Roussopoulos N, Kotidis Y (2002) Dwarf: Shrinking the petacube. In: Proceedings of the 2002 ACM SIGMOD, pp. 464–475
27. Barbara D, Sullivan M (1998) Quasi-Cubes: a space-efficient way to support approximate multidimensional databases. Technical report, ISSE George Mason University
28. Vitter JS, Wang M, Iyer B (1998) Data cube approximation and histograms via wavelets. In: Proceedings of the Seventh International Conference on Information and Knowledge Management, pp. 96–104
29. Kim A, Lakshmanan LVS, Srivastava D (2020) Summarizing hierarchical multidimensional data. In: 36th IEEE International Conference on Data Engineering , ICDE 2020, Dallas, TX, USA, April 20-24, 2020, pp. 877–888
30. You J, Xi J, Zhang P, Chen H (2008) A parallel algorithm for closed cube computation. In: Seventh IEEE/ACIS International Conference on Computer and Information Science (ICIS 2008), pp. 95–99
31. Nandi A, Yu C, Bohannon P, Ramakrishnan R (2011) Distributed cube materialization on holistic measures. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 183–194
32. Lee S, Kang SH, Kim J, Yu EJ (2019) Scalable distributed data cube computation for large-scale multidimensional data analysis on a Spark cluster. Clust Comput 22(1):2063–2087
33. Wang H, Wang Z, Li N, Kong X (2019) Efficient OLAP algorithms on GPU-accelerated Hadoop clusters. Distrib Parallel Databases 37(4):507–542
34. Lin CX, Ding B, Han J, Zhu F, Zhao B (2008) Text cube: Computing IR measures for multidimensional text database analysis. In: 2008 IEEE 8th International Conference on Data Mining, pp. 905–910
35. Qu Q, Zhu F, Yan X, Han J, Philip SY, Li H (2011) Efficient topological OLAP on information networks. In: International Conference on Database Systems for Advanced Applications, pp. 389–403
36. Heine F, Rohde M (2017) PopUp-Cubing: An algorithm to efficiently use iceberg cubes in data streams. In: Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, pp. 11–20
37. Chandra AK, Merlin PM (1977) Optimal implementation of conjunctive queries in relational data bases. In: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, pp. 77–90
38. Aho AV, Sagiv Y, Ullman JD (1979) Equivalences among relational expressions. SIAM J Comput 8(2):218–246
39. Ioannidis YE, Ramakrishnan R (1995) Containment of conjunctive queries: beyond relations as sets. ACM Trans Database Syst 20(3):288–324
40. Cohen S, Nutt W, Sagiv Y (2003) Containment of aggregate queries. In: International Conference on Database Theory, pp. 111–125