μ-join: Efficient Join with Versioned Dimension Tables

Mika Takata(✉), Kazuo Goda, and Masaru Kitsuregawa

The University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo, Japan
{mtakata,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. The star schema is composed of two types of tables; fact tables record business process, whereas dimension tables store description of business resources and contexts that are referenced from the fact tables. Analytical business queries join both tables to provide and verify business findings. Business resources and contexts are not necessarily constant; the dimension table may be updated at times. Versioning preserves every version of a dimension record and allows a fact record to reference its associated version of the dimension record correctly. However, major existing versioning practices (utilizing a binary join operator and a union operator) cause processing redundancy in queries joining a fact table and a dimension table. This paper proposes μ-join, an extended join operator that directly accepts a fact table and an arbitrary number of dimension tables, and presents that this operator reduces the redundancy and speeds up fact-dimension joins queries. Our experiment demonstrates that μ-join offers speedup using the synthetic dataset (up to 71.7%).

Keywords: Relational database · Join · Dimension table

1 Introduction

Versioning dimension tables is a widely employed practice to offer referential integrity in the star schema [12]. The star schema is a popular approach for organizing business data into relational database. The schema is composed of two types of tables. Fact tables record business process, whereas dimension tables store description of business resources and contexts. Analytical business queries join both tables to provide and verify business findings. Business resources and contexts are not necessarily constant. Suppose that database records a company's sales history. Daily sales events are recorded into a fact table, whereas product names and prices are stored in a dimension table. The company may update its business resources or contexts, for example, by launching new products, discontinuing existing products, and changing products names or prices. These updates are not necessarily frequent, but they may actually happen; the updates of business resources and contexts are described in the dimension table.

A. Bhattacharya et al. (Eds.): DASFAA 2022, LNCS 13245, pp. 88–95, 2022.
https://doi.org/10.1007/978-3-031-00123-9_6

Versioning preserves every version of a dimension record and allows a fact record to reference its associated version of the dimension record correctly. Existing versioning practices using a binary join and a union can answer queries joining a fact table and a versioned dimension table. One major practice joins a fact table and multiple versions of a dimension table one by one and then eliminates redundant tuples. Another practice merges multiple versions of a dimension table, joins it with the fact table, and then eliminates redundant tuples if necessary. These practices do not necessarily offer optimized processing, but rather induce inefficient processing.

This paper proposes, μ-join, an extended join operator, which can directly join a fact table and multiple versions of a dimension table by considering the join condition on a join key and a version compatibility. The μ-join operator allows database engines to evaluate the join conditions at an early phase and to reduce the version-related complication. Thus, the database engine potentially improves the join performance. This paper presents our experiments to clarify the benefit of μ-join in terms of query response time using a synthetic dataset. To our knowledge, similar attempts have not been reported in literature.

The remainder of this paper is organized as follows. Section 2 describes versioned dimension tables and their problem. Section 3 defines a version-aware fact-dimension join operation and presents the μ-join operator. Section 4 presents the experimental study. Section 5 shows related work and Sect. 6 concludes the paper.

2 Join with Multiple Versions of Dimension Tables

Let us exhibit another example of a versioned dimension table, which has motivated this study. Figure 1 presents an example database of Japanese public healthcare insurance claims. Japan has employed the universal service policy; all the certified healthcare services necessary to the citizens are basically covered by the public healthcare insurance system. When providing an insured patient with healthcare services, healthcare service providers (e.g., hospitals) are supposed to submit an insurance claim to request the compensation of the expense to the public healthcare insurers. Let us think about the design of database for managing the insurance claims. One typical solution is to record insurance claims in a fact table (S in the figure). Each claim contains the description of diagnosed diseases, which are coded in the standardized format. Such descriptive information of disease codes is to be stored in a separate dimension table. The disease code system does not remain unchanged. Rather, it may change at times. Figure 1 presents a partial portion of the real Japanese disease code standard, indicating that the disease code system changed twice from June, 2020 to October, 2021. For example, the disease code indicating Apparent strabismus changed from 8831256 to 3789010 after January, 2021. When healthcare researchers analyze the database, they need to interpret each insurance claim by referencing an appropriate disease description record according to the recorded disease code and the claiming date. Thus, the database must keep all historical

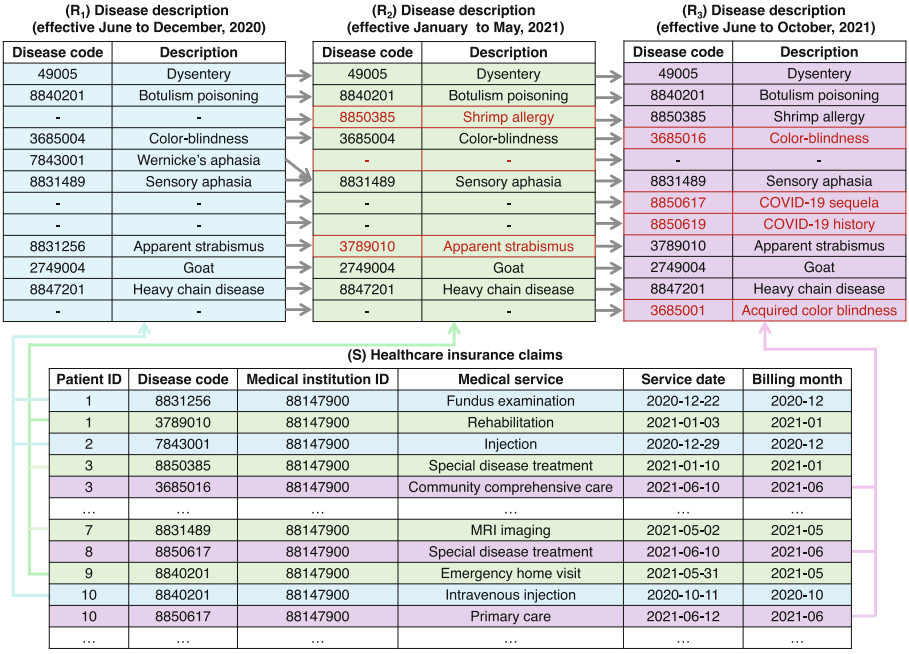


Fig. 1. An example of a fact table and versioned dimension tables

updates of the dimension tables (R_1, R_2, and R_3 in the figure) for answering such queries. This complication is not limited to disease codes. Insurance claims contain more information such as medical treatments and medicinal drugs, the descriptive information of which change every month, in order to reflect the evolution of medical science and industry.

Versioning preserves every version of a dimension record and allows a fact record to reference its associated version of the dimension record correctly. Unfortunately, most current database management systems do not support such dimension versioning explicitly. Instead, implementing the dimension versioning on the relational schema is a major engineering approach. They can be roughly grouped into three policies. *Snapshot versioning* stores every entire version of a dimension table as a separate table. *Differential versioning* stores only the difference of every version from the first version as a separate table. *Incremental versioning* stores only the difference of every version from the previous version as a separate table. Snapshot versioning is relatively simple, but it consumes redundant space and may cause additional processing due to tuple redundancy. Incremental versioning only consumes smaller space, but it is complicated, needing additional processing to reproduce a tuple of a specific version.

Analytical queries often performed across a fact table and such a versioned dimension table. Those queries must be written to guarantee that each fact record can reference its associated version of the dimension record. One typical

practice uses a binary join operation to join a fact table and every version of a dimension table, merges the join results, and then eliminates non-effective and redundant tuples. This practice often causes inefficient processing due to redundant work and elimination work. Another practice merges all possible versions of a dimension table, joins the merged dimension table, and then eliminates non-effective and redundant tuples. This practice do not necessarily offer optimized processing, but it is likely to induce inefficient processing.

3 The μ-join Operator

In this section, we introduce a new database operator, named μ-join. The μ-join operator extends the existing binary join and it can directly join a fact table and multiple versions of a dimension table by considering the join condition on a join key and a version compatibility. Being implementing this new operator, database engines can efficiently perform version-aware fact-dimension join queries.

Version-Aware Fact-Dimension Join Operation. First, we define a versioned dimension table and version-aware fact-dimension join operation.

Definition 1 (Versioned dimension table). Let $\check{R}$ be a versioned dimension table, having $<\check{R}>$ versions, where the k-th version $\check{R}_k$ is defined as $\check{R}_k := \{r | r \in \check{R} \wedge e_k(r)\}$, where $e_k(r)$ is true if and only if a tuple r is effective at a point of a version number k $(0 \leq k < <\check{R}>)$.

Definition 2 (Snapshot versioning). When snapshot versioning is deployed, each version $\check{R}_k$ is directly stored as a separate table $R_k^{(s)}$ as $R_k^{(s)} \leftarrow \check{R}_k$.

Definition 3 (Differential versioning). When differential versioning is deployed, the first version $\check{R}_0$ is directly stored as a separated table $R_0^{(d)}$ and the difference of each version $\check{R}_k$ from the first version $\check{R}_0$ is stored as a separate table $R_k^{(d)}$ in the database as $R_0^{(d)} \leftarrow \check{R}_0$, $R_k^{(d)} \leftarrow \check{R}_k \ominus \check{R}_0 (k > 0)$, where the operator $\ominus$ produces the difference of a left-hand table from a right-hand table as a table[1].

Definition 4 (Incremental versioning). When incremental versioning is deployed, the first version $\check{R}_0$ is directly stored as a separate table $R_0^{(i)}$ and the difference of each version $\check{R}_k$ from the first version $\check{R}_0$ is stored as a separate table $R_k^{(i)}$ in the database as $R_0^{(i)} \leftarrow \check{R}_0$, $R_k^{(i)} \leftarrow \check{R}_k \ominus \check{R}_{k-1} (k > 0)$.

Next, we define version-aware fact-dimension join operation that can be performed over a fact table S and a versioned dimension table $\check{R}$.

[1] Assuming that a record given in a right-hand table is deleted in a left-hand table, the operator $\ominus$ returns the deleted record with the deleted flag. This is analogous to the minus operator in the normal arithmetic system. The similar technique is widely deployed in database logging [8]. In our implementation, the deleted flag is stored in a separated attribute.

Definition 5 (Version-aware fact-dimension join operation). A version-aware fact-dimension join operation $S \bowtie_{v,Y=X} \check{R}$ of a fact table S and a versioned dimension table $\check{R}$ is defined as $S \bowtie_{v,Y=X} \check{R} := \bigcup_{0 \leq k < \langle \check{R} \rangle} \{s \cup r | s \in S \wedge r \in \check{R}_k \wedge s_Y = r_X \wedge v(s) = k\}$, where an attribute set Y is a foreign key of S, another attribute set X is a primary key of $\check{R}$ and a function $v(s)$ return a version number indicating a version of the dimension table $\check{R}$ with which a tuple s of the fact table S is associated. The equation $Y = X$ denotes a join key condition and the equation $v(s) = k$ denotes a version compatibility condition.

μ-join Operator. The version-aware fact-dimension join operation presented in Definition 5 can be expressed by a combination of multiple binary join operations, a union operation, and a selection operation. Thus, it can be performed on conventional database engines. However, such practices are likely to cause processing inefficiency as discussed in Sect. 2.

This paper proposes *μ-join*, a new database operator to be implemented in database engines. The μ-join operator extends the existing binary join. This operator directly accepts a fact table S and multiple versions of a dimension table $\check{R}_0, \check{R}_1, \cdots, \check{R}_{\langle \check{R} \rangle - 1}$ to join them by considering the join condition on a join key and a version compatibility according to Definition 5. The native implementation of the μ-join operator allows the database engine to evaluate the join key condition $s_Y = r_X$ and the version compatibility condition $v(s) = k$ at an early phase to reduce the redundant tuple processing that is likely to be imposed by the binary-join practice.

In addition, the database engine is allowed to build a *version index* data structure to identify *target* relational tables (storing a versioned dimension table) with which each incoming fact tuple s joins. This is beneficial for differential versioning and incremental versioning, which induces complexity in the identification of target relational tables. Suppose a fact tuple s has an apparent version k, meaning that the tuple s is supposed to join with $\check{R}_k$. In snapshot versioning, obviously the target dimension tuple with which the tuple s joins is stored in the dimension table $R_k^{(s)}$. However, the target dimension tuple with which the tuple s joins is stored in any of $R_0^{(d)}$ and $R_k^{(d)}$ in differential versioning or $R_0^{(i)}, \cdots, R_k^{(i)}$ in incremental versioning. The version index data structure accepts a primary key value x (for the attribute X) and an apparent version number k as input, and returns a version number indicating a target relational table storing a versioned dimension table with which a fact tuple having x and k is associated. The database engine is allowed to efficiently identify the target relational table storing a versioned dimension table for each incoming fact tuple, thus improving the efficiency of the join processing.

In the hash-based implementation of μ-join operator, hash tables are built from target relational tables storing a versioned dimension table in the first phase in similar to hash binary joins. At the same time, the version index data structure is built from the target relational tables. Building the version index data structure does not incur major performance overhead because dimension tables are often much smaller than fact tables and the majority part of the version index building is shared with the normal hash table building. In the second phase,

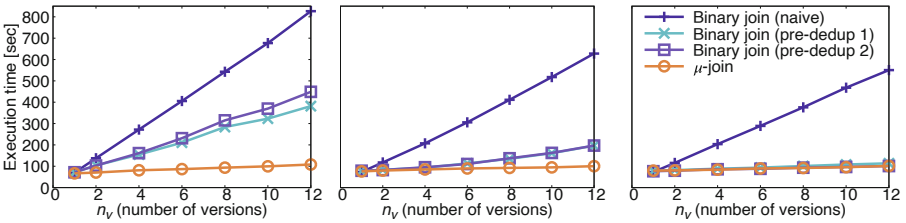
a fact table is scanned; each fact tuple probes in the version index data structure to identify the target table, and then probes in the identified target table. Thus, the μ-join operator offers an opportunity for database engines to improve the processing efficiency for the version-aware fact-dimension join operation. Also, μ-join is so intuitive that it can encapsulate the logic complication of version management in the database engine. That would potentially relieve the database designer's effort.

4 Evaluation

We present our experiment to clarify the benefit of the μ-join operator in terms of query response time. We revised the TPC-H data generator (dbgen) so that it could produce versioned dimension tables. The revised generator was enabled to produce multiple versions of dimension tables, where every version of a dimension table contained different tuple content[2] from its previous version, according to n_v ($n_v \geq 1$) that specifies the number of versions to be generated for each dimension table. The first version is identical to the original dimension table according to the TPC-H specification. In addition, the revised generator appended a version attribute, for each fact table, indicating a dimension version with which each fact tuple is supposed to join. For a given fact table, we divided the date attribute (e.g., O_ORDERDATE) space into n_v periods and determined a version of each fact tuple based the period with which the date value is associated. We generated the TPC-H dataset with versioned dimension tables organized with three version management policies.

For comparison, we tested the query execution according to the conventional binary-join practices and the new μ-join operator. **Binary join (naive)** performs the conventional binary-join operator to join a fact table and each version of a dimension table, merges the join results, and eliminates non-effective or redundant tuples. **Binary join (pre-dedup 1)** merges all possible versions of a dimension table, performs the conventional binary-join operator to join a fact table and the merged dimension table, and eliminates non-effective or redundant tuples. **Binary join (pre-dedup 2)** merges all possible versions of a dimension table, eliminates redundant dimension tuples as much as possible, performs the conventional binary-join operator to join a fact table and the merged dimension table, and eliminates non-effective or redundant tuples. ***μ*-join** performs the μ-join operator to directory join a fact tuple and all possible versions of a dimension table, without generating any non-effective or redundant tuples. We implemented these execution cases in the same code base from scratch and measured the average execution time of five trials for each test. We only employed the on-core hash join algorithm (accommodating an entire hash table in memory) rather than the nested-loop join algorithm without any selection predicates. The query execution was single-threaded and conducted on Linux with two Intel

[2] We only updated a non-key attribute for each dimension table. For example, we generated a new version of PART by shifting the value of P_RETAILPRICE randomly within the $\pm 5\%$ range from its previous version.



(a) Snapshot versioning (b) Differential versioning (c) Incremental versioning

Fig. 2. Evaluation on different numbers of versions

Xeon two processors (56 processing cores in total running as 2.60 GHz) and 96GB DRAM with 31TB RAID-6 storage by twenty-four disks.

Figure 2 presents the execution performance of a query joining LINEITEM and versioned PART on different numbers of dimension versions. Figure 2(a) shows μ-join speeds up a fact-dimension (w/two versions) join up to 31.7% from the best case of binary-join practices on snapshot versioning. As more versions are involved, μ-join achieves higher speedups up to 71.7% for twelve versions. Figure 2(b) shows μ-join consistently achieves speedups up to 2.89% for two versions and 48.9% for twelve versions on differential versioning. Figure 2(c) shows μ-join performs comparably with the best case of binary-join practices on incremental versioning. These results indicate that μ-join achieves significant speedups by evaluating the join key condition and the version compatibility at an early phase and by using the version index data structure to reduce the redundant tuple processing.

5 Related Work

Various join operator was explored for improving analytical query performance on database such as hash join, merge join, and nested loop join [1,2,13,15,16]. As a data structure, the columnar-based system was explored for statistical process to enhance analytical query performance [9,14,17]. To evolve analytical efficiency on database, many operations were implemented such as min, max, and group-by [7]. Stored procedures were explored to incorporate a user's specific requirements into database systems [6,7,18]. Sql-like operators were extended by the standard SQL to allow analysts easily do certain types of analytics [3–5,10]. Cohort analysis operation was attempted to integrate complicated processing into an operator based on requirements from real world [11]. Although such operators have been studied, join across a fact table and multiple versions of a dimension table has not been studied yet in literature to our best knowledge.

6 Conclusion

This paper proposes μ-join, an extended join operator that directly accepts a fact table and an arbitrary number of dimension tables, and presents that this

operator speeds up fact-dimension joins queries. Our experiment demonstrates that μ-join offers speedup up to 71.7% from the best case of binary-join practices for twelve versions using the synthetic dataset. There remain open problems. The query language and the query optimization should be studied to allow users to intuitively exploit μ-join on the standard query interface. We would like to consider the problems in future research.

References

1. Barber, R., et al.: Memory-efficient hash joins. PVLDB **8**(4), 353–364 (2014)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
3. Bosc, P., Dubois, D., Pivert, O., Prade, H.: Flexible queries in relational databases - the example of the division operator. Theor. Comput. Sci. **171**(1–2), 281–302 (1997)
4. Bosc, P., Pivert, O.: SQLF: a relational database language for fuzzy querying. IEEE Trans. Fuzzy Syst. **3**(1), 1–17 (1995)
5. Chatziantoniou, D., Ross, K.A.: Querying multiple features of groups in relational databases. In: VLDB, vol. 96, pp. 295–306 (1996)
6. Eisenberg, A.: New standard for stored procedures in SQL. SIGMOD Rec. **25**(4), 81–88 (1996)
7. Gray, J., et al.: Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Min. Knowl. Discov. **1**(1), 29–53 (1997)
8. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, Burlington (1993)
9. Gupta, A., et al.: Amazon redshift and the case for simpler data warehouses. In: SIGMOD, pp. 1917–1923 (2015)
10. Hosain, S., Jamil, H.: Algebraic operator support for semantic data fusion in extended SQL. In: ICCTIS, pp. 1–6. IEEE (2010)
11. Jiang, D., et al.: Cohort query processing. PVLDB **10**(1), 1–12 (2016)
12. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley, Hoboken (2011)
13. Kitsuregawa, M., Tanaka, H., Moto-Oka, T.: Application of hash to data base machine and its architecture. New Gener. Comput. **1**(1), 63–74 (1983)
14. Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing database architecture for the new bottleneck: memory access. VLDB J. **9**(3), 231–246 (2000)
15. Patel, J.M., Carey, M.J., Vernon, M.K.: Accurate modeling of the hybrid hash join algorithm. In: SIGMETRICS, pp. 56–66 (1994)
16. Shapiro, L.D.: Join processing in database systems with large main memories. ACM Trans. Database Syst. **11**(3), 239–264 (1986)
17. Stonebraker, M., et al.: C-store: a column-oriented DBMS. In: Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker, pp. 491–518. ACM/Morgan & Claypool (2019)
18. Yu, X., et al.: PushdownDB: accelerating a DBMS using S3 computation. In: ICDE, pp. 1802–1805. IEEE (2020)