A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses

George Candea
EPFL & Aster Data
George.Candea@epfl.ch

Neoklis Polyzotis
U.C. Santa Cruz
alkis@ucsc.edu

Radek Vingralek
Aster Data
Radek.Vingralek@asterdata.com

ABSTRACT

Conventional data warehouses employ the query-at-a-time model, which maps each query to a distinct physical plan. When several queries execute concurrently, this model introduces contention, because the physical plans—unaware of each other—compete for access to the underlying I/O and computation resources. As a result, while modern systems can efficiently optimize and evaluate a single complex data analysis query, their performance suffers significantly when multiple complex queries run at the same time.

We describe an augmentation of traditional query engines that improves join throughput in large-scale concurrent data warehouses. In contrast to the conventional query-at-a-time model, our approach employs a single physical plan that can share I/O, computation, and tuple storage across all in-flight join queries. We use an "always-on" pipeline of non-blocking operators, coupled with a controller that continuously examines the current query mix and performs run-time optimizations. Our design allows the query engine to scale gracefully to large data sets, provide predictable execution times, and reduce contention. In our empirical evaluation, we found that our prototype outperforms conventional commercial systems by an order of magnitude for tens to hundreds of concurrent queries.

1. INTRODUCTION

Businesses and governments rely heavily on data warehouses to store and analyze vast amounts of data; the information within is key to making sound strategic decisions. Data warehousing has recently penetrated the domains of Internet services, social networks, advertising, and product recommendation, where complex queries are used to identify behavioral patterns in users' online activities. These systems query ever increasing volumes of data—hundreds of terabytes to petabytes—and the owners of the data scramble to "monetize" it, i.e., distill the data into social or financial profit.

Unlike in the past, modern data warehouse (DW) deployments require support for many concurrent users. Commercial customers today require support for tens of concurrent queries, with some even wishing to concurrently process hundreds of reports for the same time period. Moreover, such customers desire that going from one query to several concurrent ones should not drastically increase query latency. For example, one of our large DW clients specifically asked that increasing concurrency from one query to 40 should not increase latency of any given query by more than a factor of six. Large organizations employing DWs indicate that their data warehouses will have to routinely support many hundreds of concurrent queries in the near future.

We know of no general-purpose DW system that can meet these real-world requirements today. Adding a new query can have unpredictable effects or predictably negative ones. For instance, when going from 1 to 256 concurrent queries, the query response time in a widely used commercial DBMS increases by an order of magnitude; in open-source PostgreSQL, it increases by two orders of magnitude. Queries that take hours or days to complete are no longer able to provide real-time analysis, since, depending on isolation level, they may need to operate on hours-old or days-old data.

This situation leads to "workload fear": users of the DW are prohibited from submitting ad-hoc queries and only sanctioned reports can be executed. In order to achieve better scalability, organizations break their data warehouse into smaller data marts, perform aggressive summarization, and batch query tasks. These measures, however, delay the availability of answers, restrict severely the types of queries that can be run (and consequently the richness of the information that can be extracted), and increase maintenance costs. In effect, the available data and computation resources end up being used inefficiently, preventing the organization from taking full advantage of their investment. Workload fear acts as a barrier to deploying novel applications that use the data in imaginative ways.

This phenomenon is not necessarily due to faulty designs, but merely indicates that most existing DBMSes were designed for a common case that is no longer common—workloads and data volumes, as well as hardware architectures, have changed rapidly in the past decade. Conventional DBMSes employ the query-at-a-time model, where each query is mapped to a distinct physical plan. This model introduces contention when several queries execute concurrently, as the physical plans compete in mutually-unaware fashion for access to the underlying I/O and computation resources. As a result, concurrent queries result in random I/O; for a 1-petabyte DW, even a query that touches only 0.1% of the database will still retrieve on the order of 100GB of data, thus likely performing a crippling number of random I/O operations.

Contributions. This paper introduces a query processing architecture that enables DW systems to scale to hundreds of concurrent users, issuing ad-hoc queries and receiving real-time answers. Our goal is to enable a new way of using data warehouses, in which users shed their workload fear and experiment freely with ad-hoc data analysis, drill arbitrarily deep, and broaden their queries.

More concretely, we introduce CJOIN, a physical operator that can evaluate concurrent join queries efficiently. The design of CJOIN

VLDB '09, August 24-28, 2009, Lyon, France

achieves deep sharing of both computation and resources, and it is well suited to the characteristics of modern DW platforms: star schema design, many-core systems, fast sequential scans, and large main memories. Using CJOIN as the basis, we build a query processing engine that scales gracefully to highly concurrent, dynamic workloads. The query engine employs a single physical plan that is "always on" and is optimized continuously based on run-time statistics. A new query can latch onto the single plan at any point in time, and it immediately starts sharing work with concurrent queries in the same plan. This deep, aggressive sharing is key to CJOIN's efficiency and sets it apart from prior work.

Measurements indicate that CJOIN achieves substantial improvement over state-of-the-art commercial and research systems. For 256 concurrent queries on a star schema, CJOIN outperforms major commercial and open-source systems by a factor of 10 to 100 on the Star Schema Benchmark [17]. For 32 concurrent queries, CJOIN outperforms them by up to 5x. More importantly, when going from 1 to 256 concurrent queries, CJOIN's response time increases by less than 30%, compared to over 500% for a leading commercial system.

The rest of the paper is structured as follows: §2 describes our target data warehousing setting and summarizes related work; §3 details the CJOIN operator and the new query processing engine built around it; §4 describes the deployment of CJOIN on modern DW platforms; §5 presents several extensions; §6 evaluates various aspects of performance and scalability; and §7 concludes.

2. BACKGROUND AND STATE OF THE ART

In this section, we provide background on the central problem addressed in our paper: improving support for concurrent queries in large data warehouses (§2.1). We then survey related work that has approached this or similar challenges (§2.2).

2.1 Our Target Domain

Data Warehousing Model. Below, we describe the model targeted by our solution; in §5 we show how specific assumptions behind this model can be lifted without affecting our techniques.

We consider a DW that organizes information using a star schema, which has become standard in the data warehousing industry. We assume a fact table F that is linked through foreign keys to d dimension tables $D_1, \ldots, D_d$. Following common practice, we assume that F is too large to fit in main memory and is considerably larger than the dimension tables.

The warehouse supports a workload of SQL queries, including periodic updates. Following common industrial practice, we assume that the concurrency control protocol provides snapshot isolation guarantees. In this setting, each transaction is "tagged" with a snapshot identifier, which is inherited by each query and update statement in the transaction.

We distinguish the class of SQL *star queries* that are common in DW workloads, particularly in ad-hoc data analytics. As will be seen later, this specific query structure enables us to develop efficient techniques for answering concurrent queries. Formally, a star query conforms to the following template:

SELECT $\mathcal{A}$, $Aggr_1, \ldots, Aggr_k$
FROM F, $D_{d_1}, \ldots, D_{d_n}$
WHERE $\bigwedge_{1 \le j \le n} F \bowtie D_{d_j}$ AND $\bigwedge_{1 \le j \le n} \sigma_{c_j}(D_{d_j})$ AND $\sigma_{c_0}(F)$
GROUP BY $\mathcal{B}$

Symbols $\mathcal{A}$ and $\mathcal{B}$ denote attribute sets from the referenced tables, and $Aggr_1, \ldots, Aggr_k$ are standard SQL aggregate functions, e.g., MIN, MAX, AVG. The WHERE clause is a conjunction of fact-to-dimension joins and selection predicates. A join predicate $F \bowtie D_{d_j}$ has the standard form of a key/foreign-key equi-join. A selection predicate c_j can be arbitrarily complex (e.g., contain disjunction or sub-queries), but can reference solely the tuple variable of D_{d_j} from the star query. For convenience of notation, we set c_j to TRUE if the query does not place a predicate on the corresponding table. Note that we allow for the case where $\mathcal{B} = \emptyset$ (i.e., there is no GROUP BY clause, so either $k = 0$ or $\mathcal{A} = \emptyset$) or $\mathcal{A} = \emptyset$. In the remainder of the paper, we assume the most general case, where $\mathcal{A} \neq \emptyset$, $\mathcal{B} \neq \emptyset$, and $k > 0$.

Problem Statement. We consider the problem of efficiently evaluating a large number of concurrent star queries in a single data warehouse. These queries can either be submitted directly by users, or constitute sub-plans of more complex queries.

An effective solution to this problem should yield high query throughput, as well as enable graceful degradation of query response time as the number of concurrent queries increases (i.e., avoid thrashing). This goal also implies a notion of predictability: Query response time should be determined primarily by the characteristics of the query, and not by the presence or absence of other queries executing concurrently in the system. Existing general-purpose DWs do not fare well in this setting, hence our motivation to find a solution suitable for highly concurrent data warehouses.

We emphasize that the overall workload need not be restricted solely to star queries. On the contrary, we envision a system architecture where concurrent star queries are diverted to a specialized query processor (such as the one presented in this paper) and any other SQL queries and update statements are handled using conventional infrastructure. While it is clearly desirable to support high concurrency across all types of queries, there are significant challenges even in doing so for just the subset of star queries. Moreover, this focus does not restrict the practicality of our solution, since star queries are common in DW workloads. Finally, our query evaluation techniques can be employed as sub-plans, to evaluate the star "portion" of more complex queries.

Physical Data Storage. We develop our techniques assuming that the DW employs a conventional row-store for data storage. This assumption is driven by the design of existing commercial DW solutions, including Oracle, IBM, Microsoft, Teradata, and the product within which CJOIN was developed. However, our approach can be applied equally well to different architectures. For instance, it is possible to implement CJOIN within a column store or a system employing compressed tables. We examine these cases in §5.

We do not make any specific assumptions about the physical design of the DW, such as partitioning, existence of indices, or materialized views. However, as we show in §5, CJOIN can take advantage of existing physical structures (e.g., fact table partitioning).

2.2 Related Work

Our work builds builds upon a rich body of prior research and industrial efforts in addressing the concurrency problem. We review here primarily techniques that enable work sharing, which is key in achieving high processing throughput.

Multi-Query Optimization. When a batch of queries is optimized as a unit, it becomes possible to identify common sub-expressions and generate physical plans that share the common computation [21]. This approach requires queries to be submitted in batches, which is incompatible with ad-hoc decision support queries. Moreover, common computation can be factored only within the batch of optimized queries, thus making it impossible to share work with queries that are already executing. In contrast, our approach shares work among the currently executing queries regardless of when they were

submitted and without requiring batch submission.

Work Sharing. Staged database systems [11, 12] enable work sharing at run-time through an operator-centric approach. Essentially, each physical operator acts as a mini query engine that services several concurrent queries, which in turn enables dynamic work sharing across several queries. A hash join operator, for instance, can share the build phase of a relation that participates in different hash joins in several queries. This design was shown to scale well to tens of complex queries. Our approach adopts a similar work-sharing philosophy, but customizes it for the common class of star queries. As a result, our design can scale to a substantially larger number of concurrent queries.

In the Redbrick DW [9], a shared scan operator was used to share disk I/O among multiple scan operations executing concurrently on multiple processors; however, the in-memory data and the state of other operators were not shared. Cooperative scans [24] improve data sharing across concurrent scans by dynamically scheduling queries and their data requests, taking into account current system conditions. Qiao et al. [18] have investigated shared memory scans as a specific form of work sharing in multi-core systems: by scheduling concurrent queries carefully, tuple accesses can be coordinated in the processor's cache. Our approach also leverages a form of scan sharing, but targets large warehouses, where the fact relation cannot fit in main memory. In addition to I/O, our approach also shares substantial computation across concurrent queries.

Recent work [8, 18] has investigated work-sharing techniques for the computation of aggregates on chip multiprocessors. The developed techniques essentially synchronize the execution of different aggregation operators in order to reduce contention on the hash tables used for aggregate computation. As discussed later, CJOIN can be combined with these techniques in an orthogonal fashion.

Finally, work sharing has been investigated extensively in the context of streaming database systems [3, 6, 7, 14, 15, 16]. By sharing work (or state) among continuous-query operators, a streaming DBMS can maintain a low per-tuple processing cost and thus handle a large number of continuous queries over fast streams. These techniques are specific to streaming database systems and cannot be applied directly to the environment that we target. An interesting aspect of our proposed architecture is that it incorporates elements from continuous query processing, which in turn allow us to transfer techniques from streaming databases to a DW setting. For instance, CJOIN adopts the Grouped Filter operator of Madden et al. [15], but extends it to support fact-to-dimension joins and arbitrary selection predicates; the original operator only supported range predicates on ordered attributes.

In summary, CJOIN enables a deeper form of work sharing than any prior work we know of: CJOIN employs a single plan that shares I/O, join computation, and tuple storage across all CJOIN queries that are in-flight at any given point in time.

Materialized Views. Materialized views enable explicit work sharing by caching the results of sub-expressions that appear in concurrently executing queries. The selection of materialized views is typically performed off-line, by examining a representative workload and identifying common sub-expressions [10, 20]. Capturing a representative workload is a challenging task in the context of ad-hoc decision-support queries, due to the volatility of the data and the diversity of queries. Moreover, materialized views add to the maintenance cost of the warehouse, and hence they do not offer clear advantages for the problem considered in this paper.

Constant Time Query Processing. BLINK [19] is a query processing architecture that achieves constant response time for the type of queries considered in our work. The idea is to run each query using the same plan—a single pass over a fully de-normalized, in-memory fact table—thus incurring more or less the same execution cost. CJOIN achieves a similar goal, in that it enables predictable execution times for star queries. The key differences compared to BLINK are that we do not require the database to be memory-resident, we do not require the fact table to be denormalized, and our design directly supports high query concurrency, whereas BLINK targets the execution of one query at a time.

3. THE CJOIN PIPELINE

This section details the design of CJOIN. We first provide an overview of CJOIN, using an illustrative example (§3.1) and then describe the various components in more detail (§3.2-§3.4). For clarity, we initially assume a query-only workload that references the same snapshot of the data. We then expand our discussion to mixed workloads of both queries and updates in §3.5.

Notation. In what follows, we write $\mathcal{Q}$ to denote the set of concurrent star queries that are being evaluated. We assume that each query is assigned a unique positive integer identifier, and we use Q_i to denote the query with id i. These identifiers are specific to CJOIN and can be assigned when queries are registered with the operator. Also, an identifier can be reused after a query finishes its evaluation. The maximum query id in $\mathcal{Q}$ is denoted as $maxId(\mathcal{Q})$. We note that $maxId(\mathcal{Q}) \geq |\mathcal{Q}|$ in the general case, since we do not require query identifiers to be consecutive. Moreover, we expect that $maxId(\mathcal{Q})$ is bounded by a system parameter $maxConc$ that limits the total number of concurrent queries.

We use c_{ij} to denote the selection predicate placed by Q_i on a dimension table D_j that it references. We assume that $c_{ij} \equiv TRUE$ if no explicit predicate is placed. We also define c_{i0} similarly with respect to the fact table F. Finally, we use $\mathbf{b}$ to denote a bit-vector of bounded length, and $\mathbf{b}[l]$ to denote the l-th bit. The symbol $\mathbf{0}$ denotes the bit vector with all bits set to 0.

3.1 Design Overview

CJOIN leverages the observation that star queries have a common structure: they "filter" the fact table through dimension predicates. Correspondingly, the architecture of CJOIN consists of a pipeline of components, as shown in Figure 1.

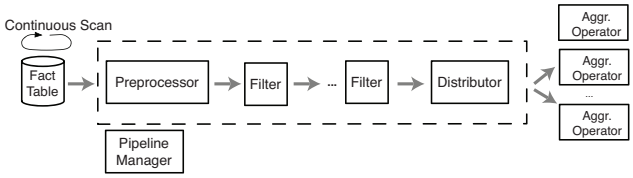

Figure 1: General architecture of the CJOIN pipeline.

The CJOIN pipeline receives its input from a continuous scan of the fact table and pipes its output to aggregation operators (either sort-based or hash-based) that compute the query results. In between, the fact tuples are processed through a sequence of Filters, one for each dimension table, where each Filter encodes the corresponding dimension predicates of *all queries* in $\mathcal{Q}$. In this way, CJOIN can share I/O and computation among all queries in $\mathcal{Q}$.

The continuous scan implies that the operator is "always on," i.e., a new query Q can be registered with the operator at any point in time. The Preprocessor marks the point in the scan where Q enters the operator and then signals the completion of Q when the scan wraps around at that same point. This design turns the fact table into a "stream" that is filtered continuously by a dynamic set of

dimension predicates.

We illustrate the operation of the pipeline and the basic ideas behind the design of CJOIN using the simple workload shown below: two star queries that join fact table F with dimension tables D_1 and D_2. The queries compute different aggregates and apply different selection predicates on D_1 and D_2.

Q_1 SELECT $Aggr_1$
FROM $F\ \tau$, $D_1\ \delta$, $D_2\ \delta'$
WHERE $\tau \bowtie \delta \bowtie \delta'$ AND $\sigma_{c_{11}}(\delta)$ AND $\sigma_{c_{12}}(\delta')$

Q_2 SELECT $Aggr_2$
FROM $F\ \tau$, $D_1\ \delta$, $D_2\ \delta'$
WHERE $\tau \bowtie \delta \bowtie \delta'$ AND $\sigma_{c_{21}}(\delta)$ AND $\sigma_{c_{22}}(\delta')$

Figure 2 shows a possible CJOIN pipeline for this workload. The following paragraphs describe the functionality of each component for this specific example.

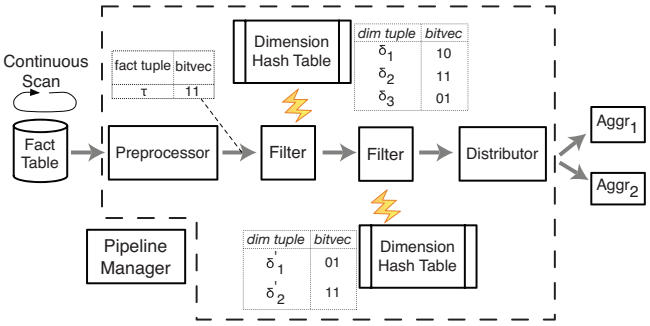

Figure 2: One possible instantiation of the CJOIN pipeline for the example queries shown above.

The *Preprocessor* receives tuples from the continuous scan and forwards them to the remainder of the pipeline. Each fact tuple τ is augmented with a bit-vector $\mathbf{b}_\tau$ that contains one bit for each query in the workload. In this example, the bit-vector consists of two bits such that $\mathbf{b}_\tau[1] = \mathbf{b}_\tau[2] = 1$. This signifies that, initially, every fact tuple is relevant for both queries Q_1 and Q_2.

The *Distributor* receives fact tuples that are relevant for at least one query in the current workload. Given a received fact tuple τ, the Distributor examines its bit-vector $\mathbf{b}_\tau$ and routes it to the aggregation operators of query Q_i if and only if $\mathbf{b}_\tau[i] = 1$.

A *Dimension hash table* stores a union of the tuples of a specific dimension table that satisfy the predicates of the current queries. In our example, say the predicates of Q_1 select exactly two tuples δ_1 and δ_2 from table D_1 and one tuple δ'_2 from D_2, while Q_2 selects tuples δ_2 and δ_3 from D_1 and tuples δ'_1 and δ'_2 from D_2. Each stored dimension tuple δ is augmented with a bit-vector $\mathbf{b}_\delta$, whose length is equal to the bit-vector $\mathbf{b}_\tau$ attached to fact tuples, with the following interpretation: $\mathbf{b}_\delta[i] = 1$ iff the dimension tuple satisfies the predicates of query Q_i. For instance, the bit-vector for tuple δ_1 is set as $\mathbf{b}_{\delta_1}[1] = 1$ and $\mathbf{b}_{\delta_1}[2] = 0$. Figure 2 illustrates the bit-vectors for all the tuples in our example.

Each *Filter* retrieves fact tuples from its input queue and probes the corresponding dimension hash table to identify the joining dimension tuples. Given a fact tuple τ, the semantics of the foreign key join ensure that there is exactly one dimension tuple δ that corresponds to the foreign key value. If δ is present in the dimension hash table, then its bit-vector $\mathbf{b}_\delta$ is combined (using bitwise AND) with the bit-vector $\mathbf{b}_\tau$ of τ. Otherwise, $\mathbf{b}_\tau$ is set to $\mathbf{0}$. The Filter forwards τ to its output only if $\mathbf{b}_\tau \neq \mathbf{0}$ after the combining (i.e., only if the tuple is still relevant to at least one query), otherwise the tuple is discarded. In this example, the first Filter outputs a tuple τ only if it joins with one of δ_1, δ_2, or δ_3. The second Filter forwards a fact tuple only if it joins with one of δ'_1 or δ'_2. Since the two Filters work in sequence, τ appears in the output of the second Filter only if its dimension values satisfy the predicates of Q_1 or Q_2.

The *Pipeline Manager* regulates the operation of the pipeline. This component is responsible for registering new queries with CJOIN and for cleaning up after registered queries finish executing. Another important function is to monitor the performance of the pipeline and to optimize it on-the-fly to maximize query throughput. For this reason and others that we mention below, it is desirable for the Pipeline Manager to operate in parallel with the main pipeline. Therefore, this component has ideally its own execution context (e.g., a separate thread or process).

Overall, the basic idea behind CJOIN is that fact tuples flow from the continuous scan to the aggregation operators, being filtered in between based on the predicates of the dimension tables. At a high level, this is similar to a conventional plan that would employ a pipeline of hash join operators to join the fact table with the dimension tables. However, CJOIN shares the fact table scan among all queries, filters a fact tuple against all queries with a single dimension table probe, and stores the union of dimension tuples selected by queries. Therefore, the fundamental difference from conventional plans is that CJOIN *evaluates all queries concurrently in a single plan that shares I/O, computation, and data*. The proposed design also differs from previous operator-centric designs (e.g., QPipe [11]) in that it takes advantage of the semantics of star queries to provide a much tighter degree of integration and sharing. For instance, QPipe would simulate two hash join operators with different state for each query, whereas in our design there is only one operator for all concurrent queries.

In this example we illustrated only one possible CJOIN pipeline for the sample workload. As we discuss later, there are other possibilities with potentially vast differences in performance. For instance, it is possible to change the order in which the Filters are applied. Another possibility is to have the Filter operators run in parallel using a variable degree of parallelism, e.g., the first Filter can employ two parallel threads while the second Filter can have just one thread. We discuss these issues in §3.4 and §4.

3.2 Query Processing

We now discuss in more detail how the CJOIN operator evaluates concurrent queries. For this part of our discussion, we assume that the workload $\mathcal{Q}$ remains fixed and that the operator uses a fixed ordering of filters. We discuss later the admission of new queries (§3.3) and the optimization of the pipeline's filter order (§3.4).

3.2.1 Dimension Hash Tables

Each dimension table D_j referenced by at least one query is mapped to a hash table HD_j, which stores those tuples of D_j that are selected by at least one query in the workload. More formally, a tuple $\delta \in D_j$ is stored in HD_j if and only if there exists a query Q_i that references D_j and δ satisfies c_{ij}. Tuple δ is also associated with a bit-vector $\mathbf{b}_\delta$ of length $maxId(\mathcal{Q})$ that determines the queries that select δ. This bit-vector is defined as follows:

$$\mathbf{b}_\delta[i] = \begin{cases} 0 & \text{if there is no query } Q_i \text{ in } \mathcal{Q} \\ 1 & \text{if } Q_i \text{ references } D_j \wedge\ \delta \text{ satisfies } c_{ij} \\ 0 & \text{if } Q_i \text{ references } D_j \wedge\ \delta \text{ does not satisfy } c_{ij} \\ 1 & \text{if } Q_i \text{ does not reference } D_j \end{cases}$$

The last case inserts an implicit $TRUE$ predicate for a query Q_i that does not reference the dimension table. The reason is that Q_i does not filter fact tuples based on D_j, so implicitly it selects all the fact tuples in D_j. The hash table also records a single complementary bitmap $\mathbf{b}_{D_j}$ defined as follows: $\mathbf{b}_{D_j}[i] = 1$ if Q_i does not reference D_j and $\mathbf{b}_{D_j}[i] = 0$ otherwise. Essentially, $\mathbf{b}_{D_j}$ is the bitmap assigned to any tuple δ that does not satisfy any of the predicates in $\mathcal{Q}$ and hence is not stored in HD_j.

By definition, HD_j stores only a subset of D_j. Each stored tuple is further augmented with a bit-vector of size $maxId(\mathcal{Q})$, which is a moderate memory overhead. Given that the dimension tables tend to grow at a much slower rate than the fact table (typically, by a logarithmic rate [17, 23]), it is reasonable to expect that the hash tables fit in main memory for modern hardware configurations. As a concrete example, TPC-DS [23] employs 2.5GB of dimension data for a 1TB warehouse; today, even a workstation-class machine can be economically equipped with 16GB of main memory.

3.2.2 Processing Fact Tuples

We consider next the details of processing a fact tuple τ through the CJOIN pipeline, starting with the Preprocessor. The Preprocessor attaches to τ a bit-vector $\mathbf{b}_\tau$ of length $maxId(\mathcal{Q})$ that traces the relevance of the tuple to different queries. This bit-vector is modified as τ is processed by Filters and it is used in the Distributor to route τ to aggregation operators.

The bit-vector is initialized based on the predicates placed on the fact table, as follows: $\mathbf{b}_\tau[i] = 1$ if $Q_i \in \mathcal{Q}_i \wedge\ \tau$ satisfies c_{i0} (i.e., the selection predicate on the fact table) and $\mathbf{b}_\tau[i] = 0$ otherwise. After $\mathbf{b}_\tau$ is initialized, the Preprocessor forwards it to its output queue if $\mathbf{b}_\tau \neq \mathbf{0}$. In the opposite case, τ can be safely dropped from further processing, as it is guaranteed to not belong to the output of any query in $\mathcal{Q}$. Computing $\mathbf{b}_\tau$ involves evaluating a set of predicates on τ, and thus it is necessary to employ an efficient evaluation mechanism to ensure that the Preprocessor does not become the bottleneck. This issue, however, is less crucial in practice, since most queries place predicates solely on dimension tables.

Tuple τ passes next through the sequence of Filters. Consider one such filter, corresponding to dimension table D_j. Let δ be the joining dimension tuple for τ. The Filter probes HD_j using the foreign key of τ and eventually computes (as explained in the next paragraph) a "filtering bit-vector" denoted by $\mathbf{b}_{\tau \bowtie HD_j}$, which reflects the subset of queries that select δ through their dimension predicates. The Filter thus joins τ with D_j with respect to *all* queries in the workload by performing a *single* probe to HD_j. Subsequently, $\mathbf{b}_\tau$ is bitwise ANDed with $\mathbf{b}_{\tau \bowtie HD_j}$. If this updated $\mathbf{b}_\tau$ vector is $\mathbf{0}$, then the fact tuple can safely be dropped from further processing, since it will not belong to the output of any query in $\mathcal{Q}$; otherwise it is passed to the output of the Filter. As an optimization, it is possible to avoid completely the probing of HD_j by checking first whether $\mathbf{b}_\tau$ AND $\neg\mathbf{b}_{D_j}$ is $\mathbf{0}$ (i.e., D_j does not appear in any query Q_i to which τ is relevant). In this case, τ is not relevant to any queries that reference HD_j and can be simply forwarded to the next Filter.

The filtering bit-vector is computed as follows: if the probe finds δ in HD_j then $\mathbf{b}_{\tau \bowtie HD_j} = \mathbf{b}_\delta$, similar to the example of Figure 2; otherwise, $\mathbf{b}_{\tau \bowtie HD_j}$ is set to $\mathbf{b}_{D_j}$, the bit-vector of any tuple that is not stored in HD_j. Given the definitions of $\mathbf{b}_\delta$ and $\mathbf{b}_{D_j}$, we can assert the following key property for the filtering bit-vector: $\mathbf{b}_{\tau \bowtie HD_j}[i] = 1$ if and only if either Q_i references D_j and δ is selected by Q_i, or Q_i does not reference table D_j. This property ensures that $\mathbf{b}_\tau$ AND $\mathbf{b}_{\tau \bowtie HD_j}$ results in a bit-vector that reflects accurately the relevance of τ to workload queries up to this point. This can be stated formally with the following invariant:

CJOIN **Filtering Invariant** *Let D_{d_1},..., D_{d_m} be the dimension tables corresponding to the first m Filters in the* CJOIN *pipeline, $m \geq 1$. If a tuple τ appears in the output of the Filter corresponding to D_{d_m}, then we have $\mathbf{b}_\tau[i] = 1$ if and only if $Q_i \in \mathcal{Q}$ and τ satisfies the predicates of Q_i on the fact table and τ joins with those dimension tuples in $\{D_{d_1}, \ldots, D_{d_m}\}$ that also satisfy the predicates of Q_i.*

Tuple τ eventually reaches the Distributor if its bit-vector is non-zero after passing through all the Filters. Given that the Filters cover all the dimension tables referenced in the current workload, the invariant guarantees that $\mathbf{b}_\tau[i] = 1$ if and only if τ satisfies all the selection and join predicates of Q_i.

The Distributor routes τ to the aggregation operator of each query Q_i for which $\mathbf{b}_\tau[i] = 1$. The aggregation operator can directly extract any needed fact table attributes from τ. If the operator needs to access the attributes on some dimension D_j, then it can use the foreign key in τ to probe for the joining dimension tuple. A more efficient alternative is to attach to τ memory pointers to the joining dimension tuples as it is processed by the Filters. Specifically, let δ be a tuple of D_j that joins to τ and assume that Q_i references D_j. Based on our definition of HD_j, it is possible to show that δ is in HD_j when τ is processed through the corresponding Filter and remains in memory until τ reaches the Distributor. This makes it possible to attach to τ a pointer to δ after HD_j is probed, so that the aggregation operator can directly access all the needed information.

3.2.3 Cost of CJOIN Query Processing

The design of CJOIN has specific implications on the cost of query processing, which we discuss below.

We consider first the end-to-end processing for a single fact tuple through the CJOIN operator. Once a tuple is initialized in the Preprocessor, if K is the total number of Filters in the pipeline, then processing the fact tuple involves K probes and K bit-vector AND operations in the worst case. Since the probe and the AND operation have limited complexity, and assuming that the Preprocessor can initialize efficiently the bit-vector of the tuple, CJOIN can sustain a high throughput between the continuous scan and the aggregation operators. Moreover, the reliance on sequential scans as the sole access method allows CJOIN to scale gracefully to large data sets, without incurring the costs of creating and maintaining materialized views or indices on the fact table, or maintaining statistics.

We discuss now the cost of a single query. The response time of a query evaluated with CJOIN is dominated by the time required to loop around the continuous scan. This cost is relatively stable with respect to the total number of queries in the workload, because the I/O is shared across all queries and the cost of probing in each Filter (cost of a hash table lookup and cost of a bitwise AND) grows at a low rate with the number of queries. Thus, as long as the rate of query submission does not surpass the rate of query completion, CJOIN yields response times with low variance across different degrees of concurrency. This property is crucial if we are to scale effectively to a large number of concurrent queries.

An added bonus is that the current point in the continuous scan can serve as a reliable progress indicator for the registered queries, and it is also possible to derive an estimated time of completion based on the current processing rate of the pipeline. Both of these metrics can provide valuable feedback to users during the execution of ad-hoc analytic queries in large data warehouses.

The predictability property implies that query response time is bounded below by the cost of a full sequential scan of the fact table. Conventional physical plans for star queries are likely to have the same property; for instance, a common plan in commercial systems is a left-deep pipeline of hash joins with the fact table as the

outer relation. In principle, the large table scan can be avoided by the use of indices or materialized views, but these structures are generally considered too expensive in the DW setting, because fact table indices have a prohibitively high maintenance cost, and ad-hoc data analysis workloads may not be stable enough to identify useful views to materialize. A common method in practice is to limit queries on specific partitions of the fact table; as will be discussed in §6, this method can also be integrated in CJOIN with similar benefits. In any case, we stress that CJOIN becomes yet one more choice for the database query optimizer; it is always possible to execute queries with conventional execution plans if this is estimated to be more efficient.

3.3 Query Admission and Finalization

Up to this point, we have examined the processing of fact tuples assuming that the CJOIN pipeline has been initialized correctly with respect to a given workload. In this section, we discuss how the state of the CJOIN pipeline is updated when a new query is admitted, or when an existing query finishes its processing.

We use n to denote the id of the query in question. For a new query, n is assigned as the first unused query id in the interval $[1, maxConc]$, where $maxConc$ is the system-wide limit on the maximum number of concurrent queries. To simplify our presentation, we assume without loss of generality that $n \leq maxId(\mathcal{Q})$.

3.3.1 Admitting New Queries

The registration of Q_n is done through the Pipeline Manager, which orchestrates the update of information in the remaining components. This approach takes advantage of the fact that the Pipeline Manager executes in parallel with the CJOIN pipeline, thus minimizing the disruption in the processing of fact tuples.

The registration is performed in the Pipeline Manager thread using Algorithm 1. The first step is to update bit n of $\mathbf{b}_{D_j}$ for each dimension table that is referenced by Q_i or appears in the pipeline (line 3). Subsequently, the algorithm updates the hash tables for the dimensions referenced in the query (line 11). For each such dimension table D_j, the Pipeline Manager issues the query $\sigma_{c_{nj}}(D_j)$ and updates HD_j with the retrieved dimension tuples. If a retrieved tuple δ is not already in HD_j, then δ is inserted in HD_j and its bit-vector initialized to $\mathbf{b}_{D_j}$. We then set $\mathbf{b}_\delta[n] \leftarrow 1$ to indicate that δ is of interest to Q_n. At the end of these updates, all the dimension hash tables are up to date with respect to the workload $\mathcal{Q} \cup \{Q_n\}$.

Having updated the dimension tables, the algorithm completes the registration by installing Q_n in the Preprocessor and the Distributor. This involves several steps. First, the Pipeline Manager suspends the processing of input tuples in the Preprocessor, which stalls the pipeline (line 17). This enables the addition of new Filters in the pipeline to cover the dimension tables referenced by the query. (While new Filters are appended in the current pipeline, their placement may change as part of the run-time optimization—see §3.4.) $\mathcal{Q}$ is also updated to include Q_n, which allows bit n of the fact tuple bit-vector to be initialized correctly. Next, the first unprocessed input fact tuple, say τ, is marked as the first tuple of Q_n, so that it is possible to identify the end of processing Q_n (see next paragraph). Finally, a special "query start" control tuple τ_n that contains Q_n is appended to the output queue of the Preprocessor, and the Preprocessor is resumed. The control tuple precedes the starting tuple τ in the output stream of the Preprocessor and is forwarded without filtering through the Filters and on to the Distributor. In turn, the latter uses the information in τ_{Q_n} to set up the aggregation operators for Q_n. Since τ_{Q_n} precedes any potential results for Q_n (the pipeline preserves the order of control tuples relative to data tuples), we guarantee that the aggregation operators will not miss any relevant fact tuples.

Algorithm 1: Admitting a new query to the CJOIN pipeline.

```
   Input: Query Q_n
   Data: A list L of dimension hash tables, initially empty
1  Let D be the set of dimension tables referenced by Q_n
2  Let D' be the set of dimension tables in the pipeline
3  foreach D_j ∈ D ∪ D' do
4      if D_j is not in the pipeline then
5          Initialize HD_j and b_{D_j} based on Q ∪ {Q_n}
6          Append HD_j to L
7      else if D_j is referenced by Q_n then
8          b_{D_j}[n] = 0
9      else
10         b_{D_j}[n] = 1
11 foreach D_j ∈ D do
12     foreach δ ∈ σ_{c_nj}(D_j) do
13         if δ is not in HD_j then
14             Insert δ in HD_j
15             b_δ ← b_{D_j}
16         b_δ[n] ← 1;
17 Stall Preprocessor;
18 foreach HD_j in L do insert a Filter for HD_j
19 Q ← Q ∪ {Q_n};
20 Set start of Q_n to next tuple in Preprocessor's input
21 Append a control tuple τ_{Q_n} in Preprocessor's output
22 Resume Preprocessor
```

It is important to note that query registration occurs in the Pipeline Manager thread and thus it can proceed, up to line 17, in parallel with the processing of fact tuples through the pipeline. This ensures that other queries are minimally disrupted during the registration of Q_n. The concurrent update of the bit-vectors in dimension hash tables does not compromise the correctness of results, since the Preprocessor continues to mark each fact tuple as irrelevant to query Q_n ($\mathbf{b}_\tau[n] = 0$). Thus, even if $\mathbf{b}_\delta[n]$ is switched on for some tuple δ (line 16), it does not lead to the generation of results for Q_n until after it becomes part of the workload in line 19, because bit n in the fact tuples' bit-vectors will be 0 until that point.

3.3.2 Finalizing Queries

Query Q_n is finalized when the continuous scan wraps around the starting fact tuple τ. Upon encountering τ in its input, the Preprocessor first removes Q_n from $\mathcal{Q}$, which ensures that the bit-vector of τ (and of any subsequent tuple) will have bit n switched off. This ensures that Q_n becomes irrelevant for the filtering of fact tuples. Subsequently, the Preprocessor emits an "end of query" control tuple that precedes τ in the output stream. The control tuple is handled in a fashion similar to the query-start tuple and is forwarded through the pipeline to the Distributor, which finalizes the aggregation operators of Q_n and outputs their results. Since the control tuple precedes τ, we ensure that the aggregation operators of Q_n will not consume any fact tuple more than once.

The final step is to clear the dimension hash tables from any information pertinent to Q_n. This is handled in the Pipeline Manager thread according to Algorithm 2, which essentially reverses the updates performed when the query was admitted. This clean-up may render certain information in the hash tables useless. For instance, if for some tuple δ in HD_j we have $\delta[i] = 0$, then δ can be removed. In turn, if HD_j becomes empty, then it can be removed from the pipeline along with the corresponding Filter. Of course, the latter requires a stall of the pipeline in order to reconfigure the Filter sequence. Note that this "garbage collection" can

Algorithm 2: Removing a finished query from the pipeline.

```
Input: Query Q_n.
Data: A list L of dimension hash tables, initially empty.
Let D be the set of dimension tables referenced by Q_n ;
Let D' be the set of dimension tables in the pipeline ;
foreach D_j ∈ D' do
    b_{D_j}[n] = 1;
foreach D_j ∈ D do
    foreach δ ∈ HD_j do
        b_δ[n] ← 0;
        if b_δ = 0 then remove δ from HD_j
    if HD_j = ∅ then Append HD_j to L
if L ≠ ∅ then
    Stall pipeline;
    foreach HD_j ∈ L do remove corresponding Filter;
    Resume pipeline;
```

be done asynchronously (as long as the query identifiers are correctly handled); one could also maintain usage bits and evict the least-recently-used tuples according to memory needs.

3.3.3 Correctness

The correctness of CJOIN with respect to query finalization hinges upon two properties. First, the continuous scan returns fact tuples in the same order once resumed. This is necessary so that the Preprocessor can identify correctly when the fact table has been scanned exactly once for each query. It is reasonable to expect that this property holds for real-world systems. The second property is that, if a control tuple τ' is placed in the output queue of the Preprocessor before (respectively after) a fact tuple τ, then τ' is not processed in the Distributor after (respectively before) τ. This property guarantees that the aggregation operators of a query neither miss relevant tuples nor process them more than once. This property needs to be enforced by the implementation of the CJOIN pipeline.

3.4 Pipeline Optimization

The order of Filters in the pipeline influences performance, because it determines the expected number of probes for each fact tuple. Drawing a correspondence to the ordering of joins in a single query plan, we expect that a good order will apply the most selective Filters first, in order to drop fact tuples early. We therefore face the following optimization problem: Given a workload $\mathcal{Q}$ and a CJOIN pipeline for $\mathcal{Q}$, determine an ordering of the Filters that minimizes the expected number of probes for each fact tuple.

One complicating factor is that the selectivity of each Filter depends on the workload $\mathcal{Q}$, since a Filter encodes the join predicates of several queries on a specific dimension table. Thus, if the workload is unpredictable, as is the case with ad-hoc analytics in a data warehouse, then the optimal order might change as the query mix changes. This observation suggests an online approach to optimizing the order of Filters in the CJOIN pipeline. The idea is to monitor at run-time the selectivity of each Filter and then optimize the order based on the gathered statistics. This continuous process of monitoring and re-optimizing can be performed asynchronously inside the Pipeline Manager thread.

Previous work introduced several techniques for optimizing the execution order of relational operators on-the-fly [4, 5, 13]. In particular, the optimization of the CJOIN pipeline maps precisely to the following problem that has been investigated in the context of streaming database systems: We are given a conjunction of predicates that are applied on the tuples of an infinite stream, and the goal is to determine an order for evaluating the predicates that minimizes the expected processing cost of each stream tuple. The correspondence to CJOIN appears when viewing Filters as predicates, and the continuous scan as an infinite stream. Moreover, since each Filter has a fixed cost—one probe of the in-memory hash table and one bitwise AND operation—minimizing the expected processing cost is equivalent to minimizing the expected number of Filter probes. In our work, we employ the techniques of Babu et al. [5] to implement the run-time optimization of the Filter ordering. A detailed discussion of these techniques is beyond the scope of our paper. We do note, however, that other techniques are applicable too, such as those introduced by Liu et al. [13].

3.5 Handling Updates

Up to this point, we have considered the case of read-only transactions that reference the same data snapshot. This enables grouping all queries of these transactions in the same CJOIN operator that performs a single continuous scan of the specific snapshot. In the remainder of this section we examine adaptations of CJOIN when this assumption is relaxed, i.e., when the queries correspond to transactions with different snapshot ids. (As mentioned in §2, we assume snapshot-based isolation, since this is the norm in practice.) This scenario arises when read-only transactions are interleaved with updates, or when the same transaction contains both queries and updates. In all cases that we examine, we focus on updates that reference only the fact table. In the rare event of updates on dimension tables, we assume that the admission of new queries in CJOIN is serialized with the transaction of the update.

We consider two possibilities for adapting CJOIN to this scenario, that depend on the functionality of the continuous scan operator. The first possibility is that the continuous scan operator can return all fact tuples corresponding to the snapshots in the current query mix. This essentially requires the scan to expose the multi-version concurrency control information for the retrieved fact tuples. Then, the association of a query Q_i to a specific snapshot can be viewed as a virtual fact table predicate, and it can be evaluated by the Preprocessor over the concurrency control information of each fact tuple. The remaining CJOIN mechanism remains unchanged. Of course, the benefits of the CJOIN operator are decreased as the snapshots referenced by the transactions become disjoint, but we believe this case to be infrequent in practice.

The second possibility is when the previous functionality is not provided, i.e., the scan only returns tuples of a specific snapshot. In this case, we create several CJOIN operators, one for each snapshot that is referenced, and register queries to the respective operator. This approach could degenerate into a single plan per query, if each transaction in our workload mix referenced different snapshot ids. This, however, is an exceptionally rare event in practice.

4. CJOIN IMPLEMENTATION

In this section, we discuss the implementation of CJOIN on a multi-core system, the predominant hardware architecture in real-world deployments today and in the near future. Nevertheless, we expect CJOIN to yield significant benefits on single-core/single-CPU hardware as well.

An efficient implementation of CJOIN on a multi-core architecture requires that the operator's components (Preprocessor, Filters, and Distributor) be mapped to multiple threads, which in turn are mapped by an operating system to different processor cores. As an example, one obvious mapping is to assign each component to a different thread and then employ tuple queues to link the pipeline. However, the mapping of CJOIN components to threads must strike a balance between the degree of parallelism, the overhead of passing tuples between the threads, and the utilization of processor

caches. Passing a tuple from one thread to another requires synchronization between the threads and also results in data cache misses if the two threads execute on different cores. On the other hand, executing components in different threads improves cache locality if the internal state is read-mostly (such as the dimension hash tables) and can be partitioned among multiple threads.

Since the internal states of the Preprocessor and Distributor are frequently updated, we chose to map each to a single thread as shown in Figure 3. Filters, where the bulk of CJOIN processing happens, do not have any internal state other than the dimension hash tables, which are read-mostly. Our implementation allows for a flexible mapping of Filters to threads by collapsing multiple adjacent Filters to a *Stage* (to reduce the overhead of passing tuples between the threads) and assigning multiple threads to each Stage (to increase parallelism). This approach gives rise to the following possible configurations:

• A **vertical** configuration assigns a single Stage to each Filter, with a (potentially) different number of threads per Stage. This design favors the affinity of Stages to cores so that instruction and data locality is maximized. On the other hand, we expect a large number of data cache misses when tuples are transferred between Filters. Moreover, the problem of pipeline optimization now acquires an extra free variable: the number of threads per Stage.

• A **horizontal** configuration assigns a single Stage to the entire sequence of Filters, and all the threads are assigned to this single Stage. This implies that several copies of the Filter sequence are running in parallel (one for each thread) and accessing the same hash tables. This scheme avoids data cache misses when tuples are passed between Filters, but may incur more misses on the accesses of the hash tables, since each thread needs to access more data. Pipeline optimization involves solely ordering the Filters.

• A **hybrid** configuration employs several Stages and an assignment of threads per Stage. This configuration can strike a balance between the two extremes of a horizontal vs. vertical configuration. More concretely, the cost of tuple passing is incurred only between Stages, and each thread needs to access solely the dimension tables that exist within a Stage. However, the run-time optimization becomes more complex, as there are now three free variables: the order of Filters, the boxing of Filters in Stages, and the assignment of threads to Stages.

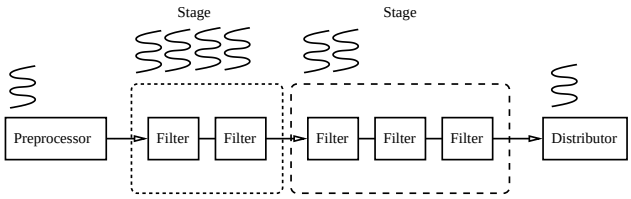

Figure 3: An example mapping of a CJOIN pipeline to threads.

Our experiments indicate that the extra parallelism resulting from using multiple Stages does not outweigh the cost of forwarding the tuples between them. Consequently, we henceforth assume the horizontal configuration, and we achieve parallelism by allocating several threads to the single Stage.

A few well-known design principles turned out to be crucial to achieving good CJOIN performance in our prototype. To reduce the overhead of thread scheduling, we wake up a consumer thread only when its input queue is almost full. Similarly, we resume the producer thread only when its output queue is almost empty. We also reduce the overhead of queue synchronization by having each thread retrieve or deposit tuples in batches, whenever possible. Finally, we reduce the cost of memory management synchronization by using a specialized allocator for fact tuples. The specialized allocator preallocates data structures for all in-flight tuples, whose number is determined based on the upper bound on the length of a tuple queue and the upper bound on the number of threads. Given the pool of preallocated tuples, the allocator reserves and releases tuples using bitmap operations, which entail a single machine instruction on most CPUs, thus being both atomic and efficient.

5. EXTENSIONS

In this section, we revisit some of the assumptions made in §2. We discuss ways in which CJOIN can be adapted to accommodate the lifting of these assumptions.

Galaxy Schemata. A galaxy schema involves several fact relations, each of which is the center of a star schema. Star queries remain common in this setting and can thus be evaluated using CJOIN, but it is also common to observe queries involving the join of several fact tables (typically two).

CJOIN can benefit the evaluation of these queries even though it was not designed for this specific case. Concretely, consider a query Q with a single fact-to-fact join predicate. By using the fact-to-fact join as a pivot, we can express the evaluation of Q as the join between the results of two star queries, say Q_a and Q_b, over the corresponding fact tables. It now becomes possible to register each Q_i with the CJOIN operator that handles the concurrent star queries on the corresponding fact table, the difference being that the Distributor pipes the results of Q_i to a fact-to-fact join operator instead of an aggregation operator. Notice that each CJOIN operator will be evaluating concurrently several star queries that participate in concurrent fact-to-fact join queries. Thus, the overall idea is to use CJOIN as a physical operator that can evaluate efficiently the "star sub-plans" of bigger query plans.

Column Stores. Column stores have been gaining traction as a scalable system architecture for large data warehouses [1, 2, 22]. It is possible to adapt CJOIN in this setting as follows: The continuous fact table scan can be realized with a continuous scan/merge of only those fact table columns that are accessed by the current query mix. Thus, CJOIN can take advantage of the columnar store in order to reduce the volume of data transferred by the scan. The other case that we need to examine is the evaluation of filter queries over dimension tables, which occurs as part of a new query registration. This case is readily handled by the column store, since CJOIN uses the existing query processing infrastructure to retrieve the resulting dimension tuples.

Compressed Tables. Data warehouses may employ compression to reduce the amount of I/O and memory bandwidth used for data transfer [1, 19]. CJOIN makes no assumptions about the physical storage of tuples, except that it is possible to evaluate predicates, extract fields, and retrieve result tuples for dimension queries. Thus, compression of tables is an orthogonal technique that can be easily incorporated in CJOIN. For instance, the continuous scan can bring in compressed tuples and decompress on-demand and on-the-fly as needed for probing the dimension hash tables. Another option is to use the partial decompression technique proposed in BLINK [19] in order to evaluate predicates efficiently on the compressed fact table.

Fact Table Partitioning. The organization of the fact table in partitions may arise naturally from the operational semantics of the DW, e.g., the fact table may be range-partitioned by a date attribute corresponding to the loading of new data. The optimizer can take ad-

vantage of this partitioning in order to limit the execution of a query to a subset of the fact table. Thus, a query that sets a range predicate on the partitioning date attribute will need to examine only a subset of the partitions. In principle, this approach can reduce significantly the response time of an individual query, but concurrent queries can still lead to random I/O, which has crippling effects on overall performance.

CJOIN can take advantage of partitioning in order to reduce the volume of data accessed by the continuous scan and also to reduce query response time. More concretely, the query registration algorithm can be modified to tag each new query with the set of partitions that it needs to scan. This set can be determined by correlating the selection predicates on the fact table with the specific partitioning scheme. The Preprocessor can then realize the continuous scan as a sequential scan of the union of identified partitions. At the end of each partition, the end-of-query control tuple can be emitted for the queries that have covered their respective set of partitions, thus allowing queries to terminate early.

Efficient Aggregate Computation. The current CJOIN design forwards the resulting tuples to aggregation operators that compute the final query results. There may be opportunities to optimize this final stage, e.g., by sharing work among aggregation operators, depending on the current query mix. This optimization is orthogonal to CJOIN and can be performed using existing techniques [8, 18].

Memory-resident Databases. The design of CJOIN was motivated by large-scale data warehouses, where the fact table is orders of magnitude larger that the available main memory. However, it is straightforward to employ CJOIN for a memory-resident data set as well. One difference is that the sharing of the continuous scan may not have as significant an effect as when the fact table resides on disk. Still, CJOIN will enable work sharing among the concurrent queries, which is important in achieving high throughput.

Indexes and Materialized Views. As discussed earlier, fact table indexes are not likely to be useful in the DW setting that we consider, due to their high maintenance cost. Similarly, the inherent volatility of ad-hoc queries limits the appearance of common patterns and hence the importance of materialized views that involve the fact table. It is more common (and affordable) for data warehouses to maintain indexes and views on dimension tables. CJOIN takes advantage of these structures transparently, since they can optimize the dimension filter queries that are part of new query registration (see also Algorithm 1).

6. EVALUATION

This section reports the results of an experimental evaluation of our CJOIN prototype. We investigate the performance characteristics of CJOIN and compare it to real-world database systems using workloads of different characteristics. In particular, we focus on the following high-level questions:

- Which is the best way to map the components of a CJOIN pipeline to CPUs? (§6.2.1)
- How does CJOIN throughput scale with increasing numbers of concurrent queries? (§6.2.2)
- How sensitive is the throughput of CJOIN to workload characteristics? (§6.2.3)
- How does the size of a data warehouse impact CJOIN's performance? (§6.2.4)

6.1 Methodology

We describe here the systems, data sets, workloads, and evaluation metrics that characterize our experiments.

6.1.1 Systems

Our CJOIN prototype is implemented as a multi-threaded process executing on top of the PostgreSQL database system. CJOIN uses PostgreSQL to issue queries over the dimension tables for the registration of new queries. The continuous scan is implemented by issuing successive `SELECT * FROM F` queries to PostgreSQL. To increase the throughput of the scan, we have implemented a fast tuple copy mechanism between PostgreSQL and CJOIN using a shared memory buffer. Our prototype supports both the horizontal (one Stage for all Filters) and vertical (one Stage per Filter) configurations described in §4.

We compare CJOIN to both a widely used commercial database system (henceforth referred to as "System X") and PostgreSQL. We tune both systems (e.g., computation of optimization statistics, allowing a high number of concurrent connections, scans using large data chunks, etc.) to ensure that the experimental workloads are executed without obvious performance problems. We have verified that both systems employ the same physical plan structure to evaluate the star queries in the experimental workloads, namely, a pipeline of hash joins that filter a single scan of the fact table. The small size of the dimension tables implies that they can be cached efficiently in main memory and hence their processing is expected to be very fast. As a result, we do not tune the physical design of any of the database systems with indices or materialized views on the dimension tables, since this would not improve query response time (we verified this claim for the experimental workloads). We also avoid using indices and views on the fact table, for the obvious reasons mentioned in previous sections. For PostgreSQL, we enable the shared-scans feature to maximize its work sharing.

Our experimental server has two quad-core Intel Xeon CPUs, with a unified 6 MB L2 cache on each CPU shared among all 4 cores and 8 GB of shared RAM. The machine has four HP 300GB 15K SAS disks, arranged in a hardware-controlled RAID-5 array.

6.1.2 Data Set and Workload

We employ the data set and queries defined in the Star Schema Benchmark (SSB) [17]. We choose this particular benchmark because it models a realistic DW scenario and targets exactly the class of queries that we consider in our work.

We generate instances of the SSB data set using the data generator supplied with the benchmark. The size of each instance is controlled by a scale factor parameter denoted as sf. A value $sf = X$ results in a data set of size X GB, with 94% of the data corresponding to the fact table. We limit the maximum value of the scale factor to 100 (i.e., a 100GB data set) to ensure the timely execution of the test workloads on our single experimental machine.

We generate workloads of star queries from the queries specified in the benchmark. Specifically, we first convert each benchmark query to a template, by substituting each range predicate in the query with an abstract range predicate, e.g., `d_year >= 1992 and d_year <= 1997` is converted to `d_year >= X and d_year <= Y`, where `X` and `Y` are variables. To create a workload query, we first sample a query template and then substitute the abstract ranges with concrete predicates based on a parameter s that controls the selectivity of the predicate. Thus, s allows us to control the number of dimension tuples that are loaded by CJOIN per query, as well as the size of the hash tables in the physical plans of PostgreSQL and System X.

Note that the original benchmark specification contains 13 queries of varying complexity. We excluded queries Q1.1, Q1.2 and Q1.3 from the generation of workload queries because they contain selection predicates on fact table attributes, and this functionality is not yet supported by our prototype. This modification does not af-

fect the usefulness of the generated workloads, since the omitted queries are the simplest ones in the SSB benchmark and the only ones that do not have a group-by clause.

6.1.3 Evaluation Metrics

We measure the performance of a system with respect to a specific workload using query throughput (in queries per hour) and the average and standard deviation of response times for each of the 10 SSB query templates. We employ the standard deviation to quantify performance stability and predictability.

For each tested system, we execute the workload using a single client and a cold cache. The degree of query concurrency is controlled by an integer parameter n, as follows: the client initially submits the first n queries of the workload in a batch, and then submits the next query in the workload whenever an outstanding query finishes. This way, there are always n queries executing concurrently. To ensure that we evaluate the steady state of each system, we measure the above metrics over queries 256...512 in the workload ($n = 256$ is the highest degree of concurrency in our experiments). The fact that we measure a fixed set of queries allows us to make meaningful comparisons across different values of n.

6.2 Experiments

This section presents a subset of the experiments that we conducted to evaluate the effectiveness of the CJOIN operator.

6.2.1 Pipeline Configuration

We begin with a comparison of the vertical and horizontal CJOIN configurations that are supported by our prototype. The vertical configuration maps each Filter to a distinct Stage, which implies that Filters work in parallel with each other. The horizontal configuration boxes all Filters in a single Stage that is assigned several threads. Thus, each thread evaluates in parallel the sequence of Filters for a subset of the fact tuples. As discussed in §4, the vertical and horizontal configurations represent the two extremes for mapping the CJOIN operator to a multi-core system.

We evaluate the performance of each configuration as we vary the total number of threads in CJOIN. Each configuration has the minimum number of threads needed for its execution; we also set an upper limit so that each CPU core does not execute more than one "active" thread. Specifically, we always set aside three cores for the PostgreSQL process and the Preprocessor and Distributor threads. This leaves five cores out of the eight available on our experimental machine, so we use this number as the upper limit for the number of Stage threads. For the horizontal configuration, all available threads go to the single Stage. The vertical configuration requires at least four threads (there are four Filters corresponding to the dimension tables in the SSB data set) and, if there is a fifth thread available, we assign it to the first Stage.

Figure 4 shows the query throughput of the two configurations as we vary the number of Stage threads. The results show clearly that the horizontal pipeline configuration consistently outperforms the vertical configuration, as long as it has more than one thread assigned to the single Stage. Upon closer inspection, we found that the overhead of passing tuples between threads, which includes L2 data cache misses and thread synchronization, outweighs the benefits gained by the parallelism of the vertical configuration. Based on these results, we focus the subsequent experiments on the horizontal configuration for the CJOIN operator.

6.2.2 Influence of Concurrency Scale

The next set of experiments evaluates the performance of the three systems as we increase n, the degree of query concurrency.

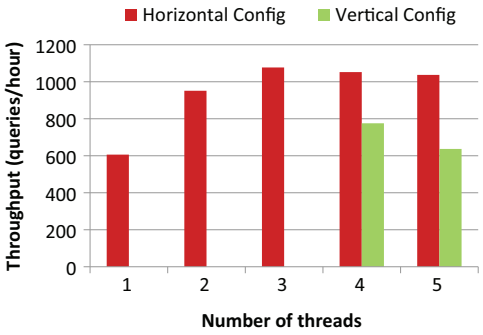

Figure 4: The effect of pipeline configuration on performance.

Ideally, a system with infinite resources would exhibit linear scaling: an increase of n by a factor k would increase throughput by the same factor. In practice, we expect a sub-linear scale-up, due to the limited resources and the interference among concurrent queries.

Figure 5 shows query throughput for the three systems as a function of n (measurements are gathered with a 100GB data set and selectivity $s = 0.01$). An immediate observation is that CJOIN delivers a significant improvement in throughput compared to System X and PostgreSQL. The improvement can be observed for $n \geq 32$ and reaches up to an order of magnitude for $n = 256$.

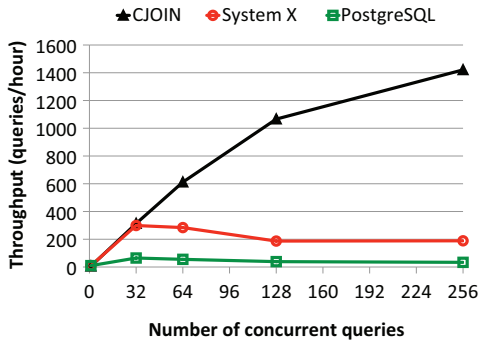

Figure 5: Query throughput scale-up with number of queries.

CJOIN achieves the ideal linear scale-up for $1 \leq n \leq 128$. Increasing n from 128 to 256 results in a sub-linear query throughput increase of 133%. We profiled the CJOIN executable and found that bitmap operations took up a large fraction of running time for this particular n, and so we believe that the sub-linear scale-up is due mostly to the specific bitmap implementation we employ. Since the efficiency of bitmap operations is crucial for CJOIN's scalability, we plan to replace the bitmap implementation.

Unlike CJOIN, the query throughputs of System X and PostgreSQL actually *decrease* when the number of concurrent queries increases past 32. As expected, this decrease is a consequence of an increased competition among all concurrently executing queries for both I/O bandwidth (for scan) and main memory (for hash tables).

We examine next the predictability of each system with respect to query response time. A system with predictable performance delivers a constant query response time independently of the number of queries that execute concurrently. To quantify this notion of predictability, we report the response times of queries generated from the template corresponding to SSB query Q4.2, which is one of the most complex queries in the benchmark (it joins with more

dimension tables than most other queries and the cardinality of its Group-By is among the largest). The results are qualitatively the same for the other templates in the benchmark.

Figure 6 shows the average response time for queries conforming to template Q4.2 as a function of n. When increasing the number of concurrent queries n from 1 to 256, the response time of System X grows by a factor of 19 and the response time of PostgreSQL grows by a factor of 66. These are precisely the undesirable performance patterns that lead to "workload fear" in existing DW platforms. CJOIN response time, on the other hand, grows by less than 30%, which is a small degradation in performance if one takes into account that the number of queries range over two orders of magnitude. Our measurements of deviation indicate that all systems deliver relatively stable query response times in steady state, although CJOIN does better: the standard deviation of response time is within 0.5% of the average for CJOIN, 5% for System X, and 9% for PostgreSQL.

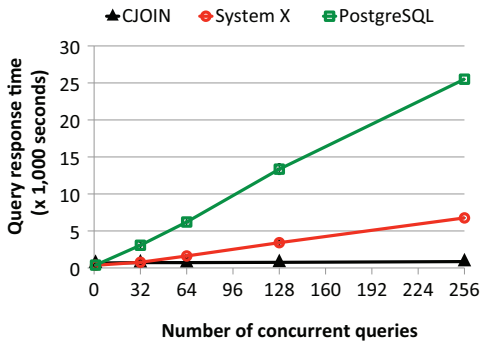

Figure 6: Predictability of query response time.

At this point, we quantify the overhead of query submission in CJOIN as we vary n. We focus again on queries matching template Q4.2 and measure the total time from the submission of the query up until the point the "start query" control tuple is inserted in the pipeline. This period represents the interval during which the submitted query competes for resources with the remainder of the pipeline, and thus it is interesting to examine its magnitude for different parameters of the workload.

Table 1 shows that the time to submit a query does not depend on the number of active queries. Moreover, the "interference" interval is small compared to the total execution time of each query. These results indicate a negligible overhead for registering a query.

n	32	64	128	256
Submission time (sec)	2.4	2.4	2.4	2.3
Response time (sec)	724.8	723.1	759.0	861.2

Table 1: Influence of concurrency on query submission time.

6.2.3 *Influence of Predicate Selectivity*

In the next set of experiments we evaluate the performance of the three systems as we increase s, the selectivity of the query template predicates. Increasing s forces all evaluated systems to access more data to answer queries. Therefore, we expect the performance to degrade at least linearly with s. However, other factors may contribute to a super-linear degradation, e.g., hash tables may not fit into L2 caches, or System X and PostgreSQL may thrash by spilling data to temporary disk files.

Figure 7 shows query throughput for all three systems as a function of s (we again use a 100 GB data set with $n = 128$ concurrent queries). First, we observe that CJOIN continues to outperform System X and PostgreSQL for all settings of s. However, we observe that the gap is reduced when $s = 10\%$, which we investigate below. Second, query throughputs of CJOIN and System X do indeed drop approximately linearly with s as expected. We cannot draw any conclusions about PostgreSQL, because we have only two data points: for $s = 10\%$, we terminated the experiment, because PostgreSQL took excessive amounts of time. Overall, we find CJOIN reacts predictably to changes in workload selectivity.

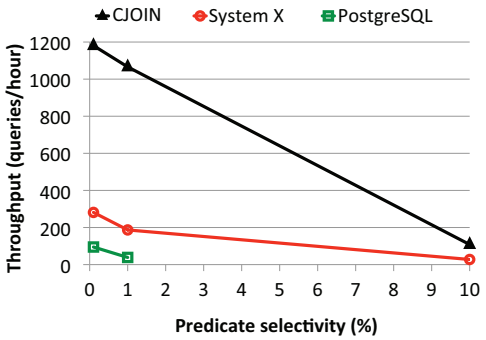

Figure 7: Influence of query selectivity on throughput.

As noted above, the performance of CJOIN decreases significantly for higher values of s. Essentially, the dimension hash tables have to hold an increased number of tuples, which has adverse effects on cache locality and hence access times. Moreover, as we explain below, the overhead of submitting new queries grows substantially, which contributes to the slow down of the operator.

Table 2 reports the overhead of new query submission for different values of s. When s increases, it is more expensive to evaluate the predicates of newly submitted queries. The dimension hash tables also grow larger, and hence it is more expensive to update them when a new query arrives. On the other hand, there are fixed costs of new query admission that do not depend on s, including the delay to submit predicate queries to the underlying PostgreSQL, to disconnect and drain the pipeline, and to update the metadata that tracks active queries in the system. As shown in the table, the factors independent of s are significant for $s \leq 1\%$, but the factors dependent on s become dominant for $s = 10\%$.

Predicate selectivity (%)	0.1	1	10
Submission time (sec)	1.6	2.4	11.6
Response time (sec)	707.2	759.0	3418.0

Table 2: Influence of predicate selectivity on query submission time.

6.2.4 *Influence of Data Scale*

In the next set of experiments, we evaluate the performance of the three systems as we increase sf, the scale factor that controls the size of the SSB data set. A scale factor $sf = \alpha$ implies a data set of α GB. Ideally, query throughput is inversely proportional to sf, since queries should take k times longer to complete on a k-times larger data set. Consequently, we expect the *normalized query throughput*, defined as a product of query throughput and sf, to remain approximately constant as sf increases.

Figure 8 shows normalized query throughput for the three systems as a function of sf (we use a workload of selectivity $s = 1\%$ and $n = 128$ concurrently executing queries). We observe that CJOIN outperforms System X for $sf \geq 1$ and PostgreSQL for all values of sf. Moreover, the performance gap increases with sf: CJOIN delivers only 85% of query throughput of System X when $sf = 1$, but outperforms System X by a factor of 6 when $sf = 100$. Similarly, CJOIN outperforms PostgreSQL by a factor of two when $sf = 1$, and by a factor of 28 when $sf = 100$.

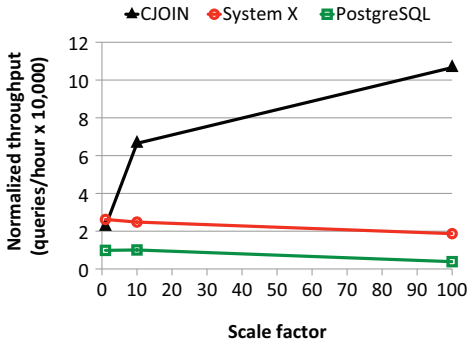

Figure 8: Influence of data scale on throughput.

Comparing the trends of query throughput, we observe that the normalized query throughput of System X and PostgreSQL *decreases* with sf, as expected, yet the normalized query throughput of CJOIN actually *increases* with sf. The explanation lies in the overhead of new query submission. As shown in Table 3, the overhead drops relative to the query response time as sf increases. The reason is twofold: (a) the fixed overhead of query submission (e.g., pipeline disconnection, or submission of predicate queries to the underlying PostgreSQL) becomes less significant as query response time grows with sf, and (b) the overhead that depends on dimension table size (e.g., evaluating dimension table predicates and updating dimension hash tables) does not grow linearly with sf, because some SSB dimension tables are fixed in size (e.g., date), and some grow logarithmically with sf (e.g., supplier and customer). Consequently, the cost of query submission becomes less significant as sf increases, and this has a positive effect on total performance.

Scale factor	1	10	100
Submission time (sec)	0.4	0.7	2.4
Response time (sec)	18.8	105.1	759.0

Table 3: Influence of data scale on query submission overhead.

7. CONCLUSIONS AND FUTURE WORK

We presented the design of CJOIN, a novel operator for the concurrent evaluation of large numbers of star-schema queries. The CJOIN design leverages sharing common parts of execution plans of multiple star-schema queries that use the same fact table. Moreover, such sharing does not require the queries to be optimized or even submitted in a batch. We presented an empirical study of CJOIN using the Star-Schema Benchmark. Our results demonstrate that CJOIN consistently outperforms both a widely used commercial database system and PostgreSQL on a variety of workloads. Furthermore, CJOIN delivers one to two orders of magnitude improvement when executing 256 concurrent queries.

8. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *Intl. Conf. on Data Engineering*, 2007.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Intl. Conf. on Data Engineering*, 2008.

[4] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2000.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2003.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. *SIGMOD Record*, 29(2), 2000.

[8] J. Cieslewicz and A. Ross, Kenneth. Adaptive aggregation on chip multiprocessors. In *Intl. Conf. on Very Large Data Bases*, 2007.

[9] P. M. Fernandez. Red Brick warehouse: A read-mostly RDBMS for open SMP platforms. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD Intl. Conf. on Management of Data*, 1996.

[11] S. Harizopoulos and A. Ailamaki. StagedDB: Designing database servers for modern hardware. *IEEE Data Eng. Bulletin*, 28(2), 2005.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *In ACM SIGMOD Intl. Conf. on Management of data*, 2005.

[13] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang. A generic flow algorithm for shared filter ordering problems. In *Symp. on Principles of Database Systems*, New York, NY, USA, 2008.

[14] Z. Liu, S. Parthasarathy, A. Ranganathan, and H. Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2008.

[15] S. Madden, M. Shah, M. Hellerstein, Joseph, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[16] A. Majumder, R. Rastogi, and S. Vanama. Scalable regular expression matching on data streams. In *Intl. Conf. on Data Engineering*, 2008.

[17] E. B. O. Patrick O'Neil and X. Chen. The Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF, 2007.

[18] L. Qiao, V. Raman, F. Reiss, P. Haas, and G. Lohman. Main-memory scan sharing for multi-core CPUs. In *Intl. Conf. on Very Large Data Bases*, 2008.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Intl. Conf. on Data Engineering*, 2008.

[20] N. Roussopoulos, C.-M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The ADMS project: Views R Us. *IEEE Data Eng. Bulletin*, 18(2), 1995.

[21] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Systems*, 13(1):23–52, 1988.

[22] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Intl. Conf. on Very Large Data Bases*, 2007.

[23] TPC benchmark DS (decision support), draft specification, revision 32. http://www.tpc.org/tpcds/spec/tpcds32.pdf.

[24] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Intl. Conf. on Very Large Data Bases*, 2007.