The VLDB Journal (2011) 20:227–248
DOI 10.1007/s00778-011-0221-2

SPECIAL ISSUE PAPER

Predictable performance and high query concurrency for data analytics

George Candea · Neoklis Polyzotis · Radek Vingralek

Received: 21 August 2010 / Accepted: 9 February 2011 / Published online: 26 February 2011

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 and thrashing, 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 and can be highly erratic when *multiple* complex queries run at the same time. We present in this paper CJOIN , a new design that substantially improves throughput in large-scale data analytics systems processing many concurrent join queries. In contrast to the conventional query-at-a-time model, our approach employs a *single* physical plan that shares I/O, computation, and tuple storage across all in-flight join queries. We use an "always on" pipeline of non-blocking operators, managed by a controller that continuously examines the current query mix and optimizes the pipeline on the fly. Our design enables data analytics engines to scale gracefully to large data sets, provide predictable execution times, and reduce contention. We implemented CJOIN as an extension to the PostgreSQL DBMS. This prototype outperforms conventional commercial systems by an order of magnitude for tens to hundreds of concurrent queries.

G. Candea
EPFL, Lausanne, Switzerland
e-mail: george.candea@epfl.ch

N. Polyzotis (✉)
University of California,
Santa Cruz, CA, USA
e-mail: alkis@ucsc.edu

R. Vingralek
Google Inc., Mountain View, Santa Clara, CA, USA
e-mail: radekv@google.com

Keywords Join · Concurrency · Warehouse · Sharing

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 analytics 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. Data analytics systems query ever increasing volumes of data—hundreds of terabytes to petabytes—and the owners of the data scramble to distill the data into social or financial profit.

Unlike in the past, today's data analytics deployments require support for many concurrent users. Commercial customers require support for tens of concurrent queries, with some even wishing to concurrently process hundreds of reports for the same time period [5]. Moreover, customers wish that the processing of several queries concurrently not increase drastically the per-query latency, relative to the single-query case. For example, one large client 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 [5]. Large organizations employing data analytics 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 data warehouse (DW) system that can meet these real-world requirements today. Adding a new query can have unpredictable effects or, at best, 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 the open-source PostgreSQL DBMS, it increases by two orders of magnitude. Yet, when queries start taking hours or days to complete, they are no longer able to provide real-time analysis, since, depending on isolation level, they may be operating on hours-old or days-old data. This situation leads to justified "workload fear," and users of the DW are prohibited from submitting ad hoc queries, with only DBA-sanctioned reports being allowed to execute.

In order to achieve better scalability, organizations break their data warehouse into smaller data marts, perform aggressive summarization, and batch query tasks. Alas, these measures 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 also 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 that maps each query to a distinct physical plan. As a result, this model introduces contention when queries execute concurrently, because the physical plans compete in mutually unaware fashion for access to the underlying I/O and computation resources. Concurrent queries therefore give rise to random I/O; given the sizes of today's DWs, the performance penalty of non-sequential access is crippling—e.g., for a 1-petabyte DW, a query that touches even only 0.1% of the database retrieves 1 TB of data using mostly random disk accesses.

This paper introduces a query processing architecture that enables DW systems to scale to hundreds of concurrent users, enabling these users to issue ad hoc queries and receive 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 as needed.

We introduce CJOIN , a new physical join operator that evaluates concurrent join queries efficiently. CJOIN 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 continuously optimized 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 commercial and open-source systems by a factor of 10 to 100 on the Star Schema Benchmark [23]. For 32 concurrent queries, CJOIN outperforms them by up to 5×. More importantly, CJOIN is significantly more predictable: when processing anywhere from 1 to 256 concurrent queries, CJOIN query response time varies by no more than 30%, compared to over 500% for a leading commercial system.

The rest of the paper is structured as follows: Sect. 2 describes in more detail the problem we address; Sect. 3-Sect. 5 describe the design of the CJOIN operator and the new query processing engine we built around it; Sect. 6 describes a prototype implementation of CJOIN on top of PostgreSQL; Sect. 7 evaluates various aspects of performance and scalability using this prototype; Sect. 8 discusses CJOIN 's path from prototype to product; Sect. 9 describes related work; and Sect. 10 concludes.

2 Assumptions and problem statement

In this section, we provide background on the central problem addressed in our paper: improving support for concurrent queries in large data warehouses. We describe the DW model we have in mind, a concrete statement of our goals, and assumptions regarding the physical data storage layer.

2.1 Data warehousing model

The model targeted by our work is described below; in Sect. 8, we show how specific assumptions behind this model can be lifted without affecting the techniques we propose.

We consider a data warehouse that organizes information using a star schema, a de facto standard in the DW industry: a fact table F is linked through foreign keys to n dimension tables $D_1, \ldots, D_n$. Following common practice, we assume that F is too large to fit in main memory, and it is considerably larger than the dimension tables.

The DW supports a workload consisting of SQL queries, including periodic updates. Following common industrial practice, we assume that the concurrency control protocol provides snapshot isolation, so each transaction is "tagged" with a snapshot identifier that is inherited by each query and update statement in the transaction. Snapshot isolation is employed by several commercial systems, including Aster Data deployments, Microsoft SQL Server, Oracle, and SQL Anywhere.

We distinguish the class of SQL *star queries*, common in DW workloads and 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 has the following template:

SELECT $\mathcal{A}, Aggr_1, \ldots, Aggr_k$
FROM F, D_1,..., D_n
WHERE $\bigwedge_{1 \le j \le n} F \bowtie D_j$ AND $\bigwedge_{1 \le j \le n} \sigma_{c_j}(D_j)$ 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 $F \bowtie D_j$ has the standard form of a key/foreign-key equi-join. A selection predicate c_j can be arbitrarily complex (e.g., contain disjunctions or sub-queries), but can reference solely the tuple variable of D_j from the star query. We allow for the case where $\mathcal{A} = \emptyset$ or $\mathcal{B} = \emptyset$, i.e., there is no GROUP BY clause, so either $k = 0$ 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 $k > 0$.

The template references all dimension tables in the database schema, but this is done solely to simplify notation. Given an arbitrary star query, we transform it to this template by adding the predicates $\sigma_{TRUE}(D_j)$ and $F \bowtie D_j$ for any non-referenced dimension table D_j. The semantics of the key/ foreign-key joins guarantee that the transformed query has exactly the same result. We stress that this transformation is conceptual and is done solely to facilitate our presentation.

2.2 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 that involve more than just joins.

What are the characteristics of an ideal solution? In theory, query throughput should scale linearly with the number of concurrently executing queries, or equivalently query response time should be independent of the number of concurrently executing queries. A more realistic goal, however, is to achieve near-linear scale-up for query throughput, with graceful degradation as the number of queries increases, i.e., avoid thrashing. This goal clearly implies a notion of predictability: Query response time should be determined primarily by the characteristics of the query not by the presence or absence of other queries executing concurrently in the system. Existing general purpose DWs do not fare well in this respect, 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, even doing so just for star queries is a significant challenge. 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.

2.3 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, such as Oracle, IBM DB2, Microsoft SQL Server, Teradata, and Aster Data's nCluster system—the product in whose context we developed CJOIN . 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 Sect. 8.

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 Sect. 8, CJOIN can take advantage of existing physical structures, such as fact table partitioning.

3 Design overview

We now provide an overview of CJOIN 's design, along with the motivation behind the high-level design decisions. For clarity, we assume a query-only workload that references the same snapshot of the data; in Sect. 4, we expand our discussion to mixed workloads containing both queries and updates.

3.1 Design principles

The goal of CJOIN is two-fold: to reduce the response time of concurrently executing joins in large-scale data warehouses and to improve the predictability of this response time, in order to eliminate workload fear.

To achieve these goals, we adopt two key design principles. First, we aim to *minimize the amount of redundant work*, which manifests in traditional DWs as unnecessary disk I/O, computation, memory copies, and synchronization. We leverage awareness of the workload (i.e., knowledge of which queries are running concurrently) to construct

a cooperative environment in which queries help each other, instead of stepping on each others' toes. This translates into a unique physical plan for all running queries, which reduces the amount of total work across all queries—Sect. 4 describes this in detail. Second, we aim for an *adaptive query processing pipeline*, which can be adjusted at low cost, in response to changes in workload and data. This is in contrast to the classical fixed physical plan approach. Adaptivity also allows for future improvements to be easily assimilated in a CJOIN implementation, without having to alter the design—Sect. 5 describes CJOIN 's adaptivity in detail.

3.2 The CJOIN pipeline

In what follows, we use $\mathcal{Q}$ to denote the set of concurrent star queries being evaluated. We assume that each query is assigned a unique non-negative integer identifier, and we use Q_i to denote the query with id i. These identifiers are specific to CJOIN and are assigned when queries are registered with the CJOIN operator. Q_i's identifier can be reused after Q_i completes. The maximum query id in $\mathcal{Q}$ is denoted as $maxId(\mathcal{Q})$. Note that $maxId(\mathcal{Q}) \geq |\mathcal{Q}|$ in the general case, since query identifiers need not be consecutive. We assume $maxId(\mathcal{Q}) < maxConc$, where the latter is a system parameter 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 also define c_{i0} similarly with respect to the fact table F. Finally, we use $\mathbf{b}$ to denote a bit vector of length at most $maxConc$ and $\mathbf{b}[l]$ to denote its l-th bit. The symbol $\mathbf{0}$ denotes the bit vector with all bits set to 0.

CJOIN leverages the observation that star queries have a common structure: they "filter" the fact table through dimension predicates. This enables us to amortize the cost of processing a fact tuple across all queries that are currently executing, thus reducing the amount of I/O and computation cost "charged" to each query. Corresponding to this observation, CJOIN 's architecture consists of a pipeline of components, as shown in Fig. 1.

The CJOIN pipeline receives its input from a continuous scan of the fact table and feeds 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 $\mathcal{Q}$. This is how CJOIN shares among all queries in $\mathcal{Q}$ both the I/O and the computation. The continuous scan keeps CJOIN "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 entered the operator, and when the scan wraps around at that same point, it signals the completion of Q. This design turns the fact table into a "stream" that is filtered continuously by a dynamic set of dimension predicates. As will be seen later, the streaming of tuples offers the opportunity to make advantageous trade-offs between memory-copy costs and synchronization.

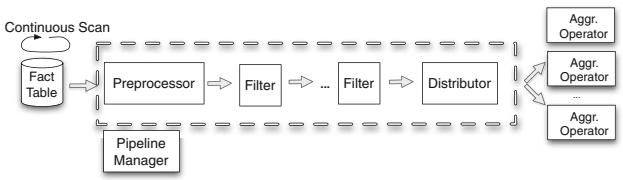

Fig. 1 High-level architecture of the CJOIN pipeline

We illustrate the operation of the pipeline and the basic ideas behind CJOIN using the following simple workload: 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.

$$
\begin{array}{ll}
 & \text{SELECT } Aggr_1 \\
Q_0 & \text{FROM } F\tau, D_1\delta, D_2\delta' \\
 & \text{WHERE } \tau \bowtie \delta \bowtie \delta' \text{ AND } \sigma_{c_{11}}(\delta) \text{ AND } \sigma_{c_{12}}(\delta') \\
 & \text{SELECT } Aggr_2 \\
Q_1 & \text{FROM } F\tau, D_1\delta, D_2\delta' \\
 & \text{WHERE } \tau \bowtie \delta \bowtie \delta' \text{ AND } \sigma_{c_{21}}(\delta) \text{ AND } \sigma_{c_{22}}(\delta')
\end{array}
$$

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

The *Preprocessor* receives tuples from the continuous scan and forwards them to the remainder of the pipeline. Each fact tuple $\tau$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 $\mathbf{b}_\tau[0] = \mathbf{b}_\tau[1] = 1$, indicating that, upon exiting the Preprocessor, every fact tuple is considered relevant to both Q_0 and Q_1, respectively.

The *Distributor* receives fact tuples that, after having crossed the pipeline, are still relevant to 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 $\mathbf{b}_\tau[i] = 1$.

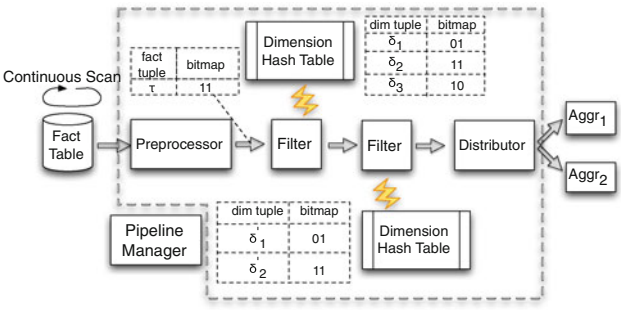

Fig. 2 One possible instantiation of the CJOIN pipeline for the two example queries shown above

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_0 select exactly two tuples δ_2 and δ_3 from table D_1 and one tuple δ'_2 from D_2, while the predicates of Q_1 select tuples δ_1 and δ_2 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, and has 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 has bits $\mathbf{b}_{\delta_1}[0] = 0$ and $\mathbf{b}_{\delta_1}[1] = 1$. Figure 2 shows the bit vectors for all 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 bitwise AND-ed with τ's bit-vector $\mathbf{b}_\tau$.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 AND-ing (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 to the Distributor 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_0 or Q_1.

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. It is also in charge of monitoring the performance of the pipeline and optimizing the pipeline 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 (and decoupled from) 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 differs from previous operator-centric designs (e.g., QPipe [15]) 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, since Q_0 and Q_1 have different selection predicates. In contrast, our design employs only one operator for both queries.

In this example, we illustrated only one possible CJOIN pipeline for the sample workload, but the CJOIN pipeline can morph into multiple shapes. As we discuss later in Sect. 5, 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. The adaptivity of the CJOIN pipeline enables easy tuning of these aspects, without requiring re-design.

3.3 Properties of the query processing pipeline

The CJOIN design 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 provide high tuple throughput from the continuous scan to the aggregation operators. 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 to providing predictable performance under high concurrency.

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. These two metrics can provide valuable feedback to users during the execution of ad hoc analytic queries in large data warehouses.

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 using fact table indices results in random I/O, and ad hoc data analytics workloads tend to not be stable enough to identify useful views to materialize. A common method used in practice is to limit queries to specific partitions of the fact table; as will be discussed in Sect. 7, this method can also be integrated in CJOIN with similar benefits. In any case, CJOIN is just another choice for the query optimizer: it is always possible to execute queries with conventional execution plans if this is estimated to be more efficient.

4 Maximizing work sharing in Cjoin

We now describe the details of each component of the CJOIN pipeline. We emphasize in this section the design elements aimed at maximizing the amount of work sharing, assuming a fixed ordering of Filters and ignoring for the time being the assignment of execution contexts to the different components. We discuss these latter two points in Sect. 5, where we review our design choices in order to enable adaptivity.

4.1 Query processing

We begin with CJOIN 's query processing logic. In this section, we assume that the workload $\mathcal{Q}$ remains fixed; admission and finalization of queries are discussed in Sect. 4.2.

4.1.1 Setting up the pipeline

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 selection predicate 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. This implies that the hash table has a bit-vector $\mathbf{b}_{D_j}$ for which $\mathbf{b}_{D_j}[i] = 1$ if Q_i does not reference D_j and $\mathbf{b}_{D_j}[i] = 0$ otherwise. This bit-vector $\mathbf{b}_{D_j}$ is implicitly assigned to any tuple δ in the dimension table that does not satisfy any of the predicates in $\mathcal{Q}$ and hence is not stored in HD_j.

The memory footprint of dimension hash tables is moderate; for example, the TPC-DS benchmark [29] employs 2.5 GB of dimension data for a 1 TB warehouse, and today even a workstation-class machine can be economically equipped with 16 GB of main memory or more. When concurrently processing a hundred queries or more, this footprint amounts to less than 25 MB of dimension table data per query, which is advantageous relative to today's systems, even if the entire dimension tables were stored in memory (note that HD_j stores only a subset of D_j, which is an optimization over storing the entire dimension tables). Since dimension tables typically grow at a much slower rate than the fact table (typically logarithmic [23,29]), it is reasonable to expect that this advantage will persist.

4.1.2 The preprocessor

We now describe the steps involved in processing a fact tuple τ through the CJOIN pipeline. First, 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: $\mathbf{b}_\tau[i] = 1$ if $Q_i \in \mathcal{Q} \wedge \tau$ satisfies c_{i0} (i.e., Q_i's 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.

4.1.3 The filters

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 $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j}$, 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* lookup in HD_j. Subsequently, $\mathbf{b}_\tau$

is bitwise AND-ed with $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j}$. 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 sometimes avoid probing HD_j by first checking whether $\mathbf{b}_\tau$ AND $\neg\mathbf{b}_{D_j}$ is $\mathbf{0}$, i.e., if 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 hash-table probe finds δ in HD_j, then $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j} = \mathbf{b}_\delta$, as in the example of Fig. 2; otherwise, $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j}$ is set to $\mathbf{b}_{D_j}$, the bit vector of tuples that are 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: $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j}[i] = 1$ 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 $\mathbf{b}_\tau \ltimes \mathbf{b}_{HD_j}$ yields 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:

Filtering invariant: Let $D_1, \ldots, D_m$ 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_m, then $\mathbf{b}_\tau[i] = 1$ 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 $\{D_1, \ldots, D_m\}$ 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 selection and join predicates of Q_i.

4.1.4 The distributor

The Distributor routes τ to the aggregation operator of each query Q_i for which $\mathbf{b}_\tau[i] = 1$ after having passed through all the dimension Filters. The aggregation operator can directly extract any needed fact table attributes from τ. If the operator needs to access the attributes of 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 the HD_j lookup, so that the aggregation operator can directly access all the needed information.

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 employ existing techniques [10,24].

4.2 Query admission and finalization

So far, we examined the processing of fact tuples assuming that the CJOIN pipeline is suitably initialized with respect to the current workload. In this section, we discuss how the CJOIN pipeline is updated when a new query is admitted or when an existing query completes its processing.

We use q to denote the id of the query in question. For a new query, q 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 $q \leq maxId(\mathcal{Q})$.

4.2.1 Admitting new queries

Query Q_q is registered 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, so the disruption to fact tuple processing is minimal.

Registration is performed in the Pipeline Manager thread using Algorithm 1. The first step is to update bit q of $\mathbf{b}_{D_j}$ for each dimension table that either is referenced by Q_q or appears in the pipeline (lines 3–10). Subsequently, the algorithm updates the hash tables for the dimensions referenced in the query (lines 11–16). For each such dimension table D_j, the Pipeline Manager issues the query $\sigma_{c_{qj}}(D_j)$ 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 $\mathbf{b}_\delta[q] \leftarrow 1$ to indicate that δ is of interest to Q_q. At the end of these updates, all the dimension hash tables are up to date with respect to the workload $\mathcal{Q} \cup \{Q_q\}$.

Having updated the dimension tables, the algorithm completes the registration by installing Q_q in the Preprocessor and the Distributor. This involves several steps. First, the Pipeline Manager suspends the processing of input tuples in the Preprocessor (line 17), which stalls the pipeline. This enables the addition of new Filters in the pipeline to cover the dimension tables referenced by the query. (Even though new Filters are appended to the current pipeline, their placement may change as part of the run-time optimization—see Sect. 5.) $\mathcal{Q}$ is also updated to include Q_q (line 19), which

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

```
   Input: Query Q_q
   Data: A list L of dimension hash tables, initially empty
 1 Let D be the set of dimension tables referenced by Q_q
 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_q}
 6         Append HD_j to L
 7     else if D_j is referenced by Q_q then
 8         b_{D_j}[q] = 0
 9     else
10         b_{D_j}[q] = 1
11 foreach D_j ∈ D do
12     foreach δ ∈ σ_{c_qj}(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[q] ← 1;
17 Stall Preprocessor;
18 foreach HD_j in L do  insert a Filter for HD_j
19 Q ← Q ∪ {Q_q};
20 Set start of Q_q to next tuple in Preprocessor's input
21 Append a control tuple τ_{Q_q} in Preprocessor's output
22 Resume Preprocessor
```

allows bit q of the fact tuple bit-vector $\mathbf{b}_\tau$ to be initialized correctly. Next, the first unprocessed input fact tuple, say $\hat{\tau}$, is marked as the first tuple of Q_q (line 20), so that it is possible to identify the end of processing Q_q (when the fact table scan wraps around $\hat{\tau}$). Finally, a special "query start" control tuple τ_{Q_q} is appended to the output queue of the Preprocessor (line 21), and the Preprocessor is resumed. The control tuple precedes the starting tuple $\hat{\tau}$ 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_q} to set up the aggregation operators for Q_q. Since τ_{Q_q} precedes any potential results for Q_q (the pipeline preserves the order of control tuples relative to data tuples), we guarantee that the aggregation operators do not miss any relevant fact tuples.

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_q. 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_q ($\mathbf{b}_\tau[q] = 0$). Thus, even if $\mathbf{b}_\delta[q]$ is switched on for some tuple δ (line 16), it does not lead to the generation of results for Q_q until after it becomes part of the workload in line 19.

4.2.2 Finalizing queries

Query Q_q is finalized when the continuous scan wraps around the starting fact tuple τ. Upon encountering τ in its input, the Preprocessor first removes Q_q from $\mathcal{Q}$, which ensures that the bit vector of τ (and of any subsequent tuple) will have bit q switched off. This ensures that Q_q 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_q and outputs their results. Since the control tuple precedes τ, we ensure that the aggregation operators of Q_q will not consume any fact tuple more than once.

The final step is to clear the dimension hash tables of any information pertinent to Q_q. 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 $\mathbf{b}_\delta = \mathbf{0}$, 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 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.

4.2.3 A note on correctness

The correctness of CJOIN with respect to query finalization hinges on two properties: First, the continuous scan must

Algorithm 2: Removing a finished query from the pipeline.

```
   Input: Query Q_q.
   Data: A list L of dimension hash tables, initially empty.
 1 Let D be the set of dimension tables referenced by Q_q ;
 2 Let D' be the set of dimension tables in the pipeline ;
 3 foreach D_j ∈ D' do
 4     b_{D_j}[q] = 1;
 5 foreach D_j ∈ D do
 6     foreach δ ∈ HD_j do
 7         b_δ[q] ← 0;
 8         if b_δ = 0 then remove δ from HD_j
 9     if HD_j = ∅ then  Append HD_j to L
10 if L ≠ ∅ then
11     Stall pipeline;
12     foreach HD_j ∈ L do remove corresponding Filter;
13     Resume pipeline;
```

return fact tuples in a consistent well-defined order, so that the Preprocessor can reliably identify 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.

4.3 Handling updates

So far, we 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 that allow us to relax this assumption for the case when queries correspond to transactions with different snapshot ids. (As explained in Sect. 2, we justifiably assume snapshot-based isolation.) This scenario arises when read-only transactions are interleaved with updates or when the same transaction contains both queries and updates. In all examined cases, we focus on updates that reference only the fact table (see below for dimension tables).

We consider two possibilities for adapting CJOIN to this scenario, depending 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 corresponds to a scan operator that 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 with the corresponding 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. Note that, even though each CJOIN operator requires a scan of the fact table, the dimension hash tables can be shared across all active operators.

Updates to the dimension tables are rare in practice, so the simplest solution is to serialize the admission of new queries in CJOIN with the dimension updates. If such updates are frequent, we can employ multi-version concurrency control in order to retrieve the right version of dimension tuples for a running CJOIN pipeline, thus avoiding the disruption due to the dimension updates.

5 Run-time adaptivity

So far, our description of CJOIN 's design focused on work sharing, assuming a fixed ordering for the Filters and a static mapping between Filters and execution contexts (i.e., processes or threads). The mapping essentially determines the degree of parallelization within the pipeline. For instance, one could assign several threads per Filter, in order to maximize utilization of available CPU cores, although this may not necessarily be the best option, as we discuss later.

Conceptually, the problem can be formulated as follows: determine the ordering of Filters and a mapping of execution contexts to Filters that maximize performance for the current workload and the current data. In this section, we extend our design to allow for run-time adjustment of these two knobs, ordering and mapping. It is important to emphasize the run-time aspect of the optimization problem—we are interested in optimizing the service rate with respect to the *current* workload and data, since new queries can attach at any point in time to CJOIN and there are no a-priori assumptions on the distribution of the workload or the distribution of updates. Our design choice is to execute the strategy for run-time adaptivity periodically inside the Pipeline Manager thread. The Pipeline Manager also becomes responsible for gathering any run-time statistics needed by the strategy.

Our performance metric is the service rate of the Filter pipeline, since it directly determines the speed of fact tuple processing and hence CJOIN 's performance (further details later). The service rate is linked to the total work performed by the Filters, which is determined in turn by the respective Filter *selectivities*. The selectivity of a specific Filter can be defined as the probability that it forwards an input tuple to its output, and it directly affects the amount of work performed by the subsequent Filters in the pipeline. Intuitively, selective Filters should appear early in the pipeline in order to reduce the volume of fact tuples that are processed by the remaining Filters; CJOIN 's run-time adaptivity is driven by this long established rule of thumb [13]. However, note that a Filter encodes the predicates of all concurrent queries, hence its selectivity is actually the average of the query-specific predicate selectivities, and hence the selectivity can change unpredictably as the workload or the data are modified. This implies that Filter selectivity has to be monitored and estimated at run time, and CJOIN must adapt its configuration to the current selectivities.

The following sections present the details of our design. We begin with basic definitions and notation and then discuss strategies for run-time adaptivity.

5.1 Definitions and notation

To simplify notation, we consider a CJOIN pipeline with a fixed set of n Filters. Let O denote a specific ordering of the Filters, such that O_i is the Filter in position i, $1 \leq i \leq n$. We use O^j to denote the prefix sequence $O_1, \ldots, O_j$. By convention, $O^{\emptyset}$ is the empty sequence.

Let τ denote a random fact tuple. We say that some Filter accepts τ if it forwards τ to its output after probing the corresponding hash table. For $1 \leq j < i \leq n$, we define $P(O_i \mid O^j)$ as the probability that Filter O_i will accept τ, given that τ is accepted by every filter in the ordering O^j. Clearly, this probability term depends on the dimension predicates encoded in O_i and O^j and on the correlation between these predicates. If Filters encode independent predicates, then $P(O_i \mid O^j) = P(O_i \mid O^{\emptyset})$, but in general, we do not expect this to be true for real-world workloads. Moreover, the probability can only be defined with respect to the current mix of queries and can change significantly when new queries arrive or existing queries are finalized.

We define a *Stage* as a non-empty sequence of Filters. Any ordering O can thus be described as a sequence of Stages $S_1, \ldots, S_m$, for $m \in [1, n]$. There are no restrictions on the subsequence corresponding to each Stage, e.g., for $m = 1$, there exists a single Stage that is exactly the ordering O, for $m = n$, each Stage has exactly one Filter, and any configuration in between is allowed. As we will see later, a Stage is the unit of assignment for execution contexts (threads of processes). Thus, a Stage with x contexts can process x fact tuples in parallel through the corresponding Filter sequence. This also means that the Filter pipeline becomes a linear queuing system with m servers, where each server corresponds to a Stage and communication is realized using tuple queues. Figure 3 captures this view of the pipeline.

Assume that Stage S_i corresponds to a sequence of m_i Filters, with $S_i[j]$ denoting the Filter at position j, $1 \leq j \leq m_i$. Accordingly, we define S^j as the Filter sequence corresponding to the concatenation of $S_1, \ldots, S_j$. We define $t(S_i)$ as the expected time to process a random fact tuple τ through S_i's sequence of Filters. Formally, let $\alpha(S_i[j])$ denote the time to probe the dimension hash table corresponding to $S_i[j]$. Then, $t(S_i)$ is computed as follows.

$$t(S_i) = \sum_{1 \leq j \leq m_i} (1 - P(S_i[j] | S^{i-1} S_i[1] \ldots S_i[j-1]) \alpha(S_i[j])$$

Here, $S^{i-1} S_i[1] \ldots S_i[j-1]$ denotes the concatenation of the Filter sequence S^{i-1} with the sequence $S_i[1], \ldots, S_i[j-1]$. Stage S_i's service rate (fact tuples/unit of time) is given by $1/t(S_i)$. We use $t(S_i, x)$ to denote the tuple-processing time if S_i is assigned x execution contexts. An ideal scale-up corresponds to $t(S_i, x) \leq t(S_i)/x$, although this is difficult to achieve because of non-linear overheads, such as cache misses or cache synchronization. A practical approach to compute $t(S_i, x)$ is to equate it to $t(S_i)\phi(S_i, x)$ and then to estimate the scale-up factor $\phi(S_i, x)$ through run-time profiling of S_i.

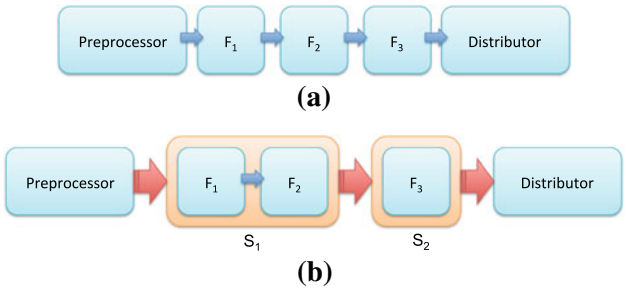

Fig. 3 An example CJOIN operator with two stages. Part (**a**) depicts the logical view of the pipeline, which consists of three Filters. Part (**b**) shows the realization of the pipeline with two stages S_1 and S_2. Each Stage is assigned disjoint execution contexts. The *thick arrows* denote the tuple queues that enable communication between different contexts. In this particular example, we assume that the Preprocessor and the Distributor are assigned their own distinct execution contexts

We can now define formally the service rate of a CJOIN pipeline, which is the objective function for our optimization problem. We are given a sequence S of m Stages and an assignment $X = x_1, \ldots, x_m$ of execution contexts to Stages. X is a *stable assignment* if $t(S_{i+1}, x_{i+1}) \leq t(S_i, x_i)/Prob(S_i \mid S^{i-1})$, i.e., the service rate of S_{i+1} exceeds the maximum output rate of S_i, for all $1 \leq i < m$. This property is derived from the basic stability condition of queuing systems. For a stable assignment X, *the service rate of the pipeline* is defined as $1/t(S_1, x_1)$, i.e., the speed of the first Stage.

5.2 CJOIN optimization for run-time adaptivity

At a high level, the problem of run-time adaptivity can be defined as computing a mapping of Filters to Stages, an ordering of Stages and of Filters within each Stage and an assignment of execution contexts to Stages that, together, maximize the service rate of the pipeline.

Definition 1 (*General adaptivity problem*) Given a set of n Filters and a total count of y execution contexts, compute a sequence of Stages $S_1, \ldots, S_m$ and a stable assignment $X = x_1, \ldots, x_m$ of execution contexts that minimizes $t(S_1, x_1)$ subject to $\sum_{1 \leq i \leq m} x_i \leq y$.

The objective function implies that we should assign more execution contexts to the first Stage in order to increase its service rate. However, the remaining Stages should be assigned

sufficient contexts to yield a stable assignment. The objective function and the stability condition are computed in terms of the service time $t(S_i, x_i)$ of each Stage, which depends in turn on the ordering of Stages, the ordering of Filters within each Stage, and on the selectivities of Filters.

The optimization problem is related to the Pipelined Set Cover problem [22], which is known to be NP-hard. Our conjecture is that the problem of general adaptivity has a similar (or even higher) complexity. Since it is unclear whether we can solve the problem of general adaptivity efficiently, we instead consider two variants that significantly restrict the solution space and hence may admit efficient solutions.

5.3 Single-Stage adaptivity

The first variant employs a single Stage that is assigned all execution contexts, i.e., we fix $m = 1$. The only free variable in this variant is the ordering of Filters within the single Stage S_1. All execution contexts are assigned to the single Stage. Hence, several copies of the Filter sequence are running in parallel (one for each context) and access 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 context needs to access more data.

The optimization problem becomes equivalent to the Pipelined Set Cover problem [22] and can thus be solved using existing techniques [6,18]. These techniques employ run-time profiling in order to estimate the current Filter selectivities and periodically reorder the Filter sequence in order to minimize $t(S_1, y)$.

5.4 n-Stage adaptivity

The second variant fixes $m = n$, i.e., each Stage corresponds to a single Filter. The free variables are the assignment of execution contexts and the ordering of Filters. Hence, Filters run in parallel to each other, and each Filter may employ parallelism to reduce its service time. This design favors the affinity of Stages to CPU cores, so that instruction and data locality are maximized. On the other hand, the transfer of tuples between Filters could incur a large number of data cache misses.

This variant is more complicated than Single-Stage adaptivity, because both the ordering of Filters and the assignment of execution contexts to Filters are free variables. We can employ a two-step approach that builds on the existing techniques for Single-Stage adaptivity: First, we determine an ordering of the Filters assuming that they all belong to a single Stage [6,18]. Second, we assign execution contexts for this specific ordering so as to achieve a stable assignment and maximize the number of contexts for the first Filter. This can be done with simple dynamic programming, by examining each suffix of the Filter sequence in order of increasing length and computing the service rate of the suffix for different values of the total number of execution contexts.

We revisit these two variants in the next section that describes our prototype CJOIN implementation. The empirical results with test workloads indicate that the Single-Stage variant provides superior performance. Essentially, n-Stage adaptivity has to pass fact tuples between the execution contexts of different Stages, which comes at a significant overhead cost. The experimental evidence, along with the simpler implementation of Single-Stage adaptivity, makes the latter the preferred approach for run-time adaptivity.

5.5 Adaptivity for heterogeneous workloads

Up to this point, our treatment of adaptivity was driven solely by the conditional selectivity of each Filter, which is essentially the average of selectivities of the dimension predicate encoded by the Filter. We now turn our attention to the important role played by workload heterogeneity.

We illustrate the concept of heterogeneity through a simple example: assume that each registered query places a single dimension predicate of low selectivity on a distinct dimension table. Furthermore, each query "selects" a disjoint set of fact tuples, i.e., a fact tuple τ can be a witness for at most one query in the mix. Let q_j be the query that places a predicate on table D_j. The hash table of the corresponding Filter will contain the dimension tuples that are relevant for q_j, and a "dense" default bitmap $\mathbf{b}_{D_j}$ that has its bits set for the remaining queries. Thus, for any ordering of the Filters, most fact tuples τ will be processed through the entire pipeline before getting dropped or forwarded to the Distributor—essentially, the service rate of the pipeline is fixed and cannot be optimized. The issue is the heterogeneity of the workload: the queries have different witnesses in the fact table and place predicates on different dimension tables. This characteristic causes most hash-table probes to return the default bitmap $\mathbf{b}_{D_j}$, whose density makes the fact tuple be highly likely to be accepted by the Filter.

We propose to employ multiple CJOIN operators in order to address the problem of workload heterogeneity. The idea is to partition the current workload in groups of similar queries, i.e., queries with similar sets of fact table witnesses, and to process each group through a separate CJOIN operator. The general idea is depicted in Fig. 4. The similarity between two queries Q_i and Q_j can be quantified as the similarity of the sets $F \ltimes \sigma_{c_{i1}}(D_1) \cdots \ltimes \sigma_{c_{in}}(D_n)$ and $F \ltimes \sigma_{c_{j1}}(D_1) \cdots \ltimes \sigma_{c_{jn}}(D_n)$, respectively, for some appropriate set-similarity metric (e.g., Jaccard similarity). In turn, set similarity can be efficiently estimated using approximate query answering techniques.

As shown in Fig. 4, the different CJOIN operators receive their input from the same continuous scan in order to

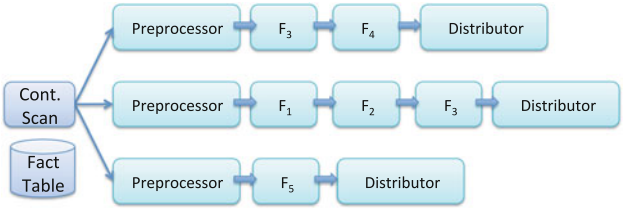

Fig. 4 An example of adaptivity through multiple CJOIN operators. Each operator has a distinct set of Filters corresponding to a relatively homogeneous partition of the workload. The operators share the same continuous scan over the fact table and can optimize their configuration independently

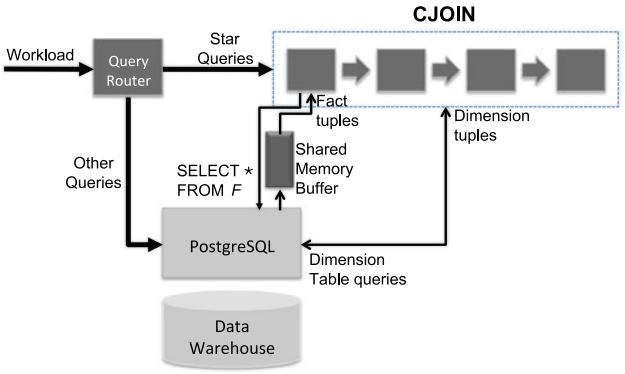

Fig. 5 Architecture of our CJOIN prototype

maximize I/O sharing. In this configuration, the total throughput is determined by the slowest operator in the group, and hence, it becomes important to optimize the maximum service time among the CJOIN operators. Let $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ denote a partition of the current workload $\mathcal{Q}$, and let $t(\mathcal{Q}_i, y_i)$ be the optimal service time of the CJOIN operator that processes $\mathcal{Q}_i$ when it is assigned y_i execution contexts. This optimal time is defined according to the General Adaptivity problem that we introduced earlier. Our goal is to compute the partitioning and assignment of contexts in order to minimize the maximum $t(\mathcal{Q}_i, y_i)$ for $1 \leq i \leq k$.

Definition 2 (Multiple-CJOIN adaptivity) Given the current workload $\mathcal{Q}$ and a total count y of execution contexts, determine a partition $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ of $\mathcal{Q}$ and an assignment $y_1, \ldots, y_k$ of contexts to minimize $\max_{1 \leq i \leq k} t(\mathcal{Q}_i, y_i)$ subject to $\sum_{1 \leq i \leq k} y_i = y$.

Multiple-CJOIN adaptivity introduces yet another dimension to the problem of run-time adaptivity, and we plan to further investigate it as part of our future work on CJOIN .

6 CJOIN Prototype

We implemented CJOIN on top of PostgreSQL. In this section, we discuss some of the interesting details of the prototype and the design choices that proved crucial in achieving good performance.

6.1 Design Goals and Constraints

Figure 5 shows an overview of the prototype. CJOIN is implemented as a multi-threaded process executing in parallel with the main PostgreSQL query engine. A query router module is responsible for routing all star queries to CJOIN , while all other queries execute inside PostgreSQL.

There are several reasons that led us to adopt this design. We create a system that supports both CJOIN-based processing and conventional query evaluation (thus, both star and non-star queries), without requiring a full integration of CJOIN as a physical query operator inside the query optimizer. At the same time, CJOIN can take advantage of the DBMS in order to evaluate dimension predicates and access fact tuples in an efficient manner. Moreover, the DBMS ensures that CJOIN will retrieve snapshot-consistent data. By implementing CJOIN as an operator external to PostgreSQL, we avoid the substantial disruption that inevitably accompanies any change to the internals of a legacy software system.

The need for high performance imposes several requirements on the implementation. Due to the pipelined design, it is crucial to achieve a high tuple transfer rate from the DBMS to CJOIN , so we need an efficient mechanism for inter-process tuple transfer. Once the fact tuples have entered the CJOIN pipeline, we need to ensure a high tuple transfer rate between the pipeline's components; given that these components may execute in different threads, we need an efficient inter-thread transfer mechanism. Finally, we need to tune carefully the pipeline configuration with respect to the schemes described in Sect. 5, e.g., determine the grouping of Filters in Stages and the assignment of threads to Stages.

6.2 Design choices and implementation details

We now show how these goals and challenges influenced the design of our CJOIN prototype.

6.2.1 Transferring dimension tuples from the DBMS

CJOIN uses the `libpq` library to connect to PostgreSQL and evaluate dimension predicates for the registration of new queries. Since such dimension table queries are infrequent and access relatively small relations, we did not seek to optimize this part further.

6.2.2 Transferring fact tuples from the DBMS

The continuous scan is implemented by issuing successive `SELECT * FROM F` queries to PostgreSQL. Our initial implementation used the `COPY` command to output fact tuples to a UNIX pipe, which operated as a queue between the continuous scan and the Preprocessor. However, we found that this design limited severely the scan's throughput. Essentially, each tuple was parsed twice—once inside PostgreSQL to write it to the pipe and once in the Preprocessor when read from the pipe—which resulted in substantial overhead. Moreover, the overhead of read/write synchronization on the UNIX pipe proved more significant than expected.

To overcome this limitation, we implemented a variant of the PostgreSQL `COPY` command that deposits unparsed binary tuples into a shared-memory queue. The new command has exactly the same syntax as `COPY`, except that the destination clause encodes the location and size of the queue in shared memory. Using this command, the raw binary tuples are inserted in the queue with little overhead, and they are retrieved and parsed exactly once inside the Preprocessor. The new design enabled a twofold increase in the throughput of the sequential scan in our test system.

The extended `COPY` command, which is accessible through the conventional SQL interface, is of general interest for the implementation of other external operators.

6.2.3 Transferring tuples in the CJOIN pipeline

We observed a significant overhead in synchronizing thread access to the shared tuple queues that link CJOIN 's components. 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 synchronization by having each thread retrieve or deposit tuples in batches, whenever possible.

6.2.4 Memory management for fact tuples

Our measurements revealed a high overhead in the allocation/deallocation of fact tuples. The initial implementation allocated new memory for each fact tuple that entered the Preprocessor and deallocated this memory when the tuple was processed by the Distributor or dropped by a Filter. Whereas memory allocation can be performed efficiently in a multi-threaded process, memory deallocation acquires a central lock inside the memory allocator and thus has a higher overhead. Combined with the large number of processed fact tuples, this resulted in a significant performance penalty.

We addressed this bottleneck by creating a custom memory allocator for fact tuples. The allocator preallocates a fixed-size pool of fact tuples when CJOIN is initialized. The size of the pool is computed based on the maximum number of in-flight fact tuples, which in turn is determined by the sizes of the queues in the CJOIN pipeline. The allocator also preallocates a bitmap that stores information on which slots are currently in use. To allocate a fact tuple, the Preprocessor simply traverses the bitmap to find a bit that can be switched on. Conversely, returning a tuple to the pool entails atomically zeroing the corresponding bit. Using this design, tuple allocation and deallocation are implemented without locks, relying solely on atomic and efficient bitmap operations that are implemented as single instructions on modern CPUs.

6.2.5 Pipeline configuration

Since the internal states of the Preprocessor and Distributor are frequently updated, we chose to devote a dedicated thread to each one of them. Of the remaining threads, some are assigned to the PostgreSQL server, one is assigned to the Pipeline Manager and the rest are assigned to Stages.

Our prototype supports both the Single-Stage and n-Stage adaptivity strategies described in Sect. 5. Recall that the n-Stage strategy maps each Filter to a distinct Stage, which implies that Filters work in parallel with each other. The Single-Stage strategy groups all Filters into 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.

We performed several experiments to evaluate the efficiency of the two strategies in the prototype implementation. Specifically, our goal was to gauge the performance of each strategy as we vary the total number of threads in CJOIN , assuming that both strategies use the same ordering for the Filters. The details of the experimental setup are given in Sect. 7. Here, we present a summary of the measurements and a discussion of the pros and cons of the two strategies.

To ensure a fair assessment, we made certain that each strategy had at least the minimum number of threads needed for its execution. Our experimental setup created pipelines of four Filters, which means that the n-Stage strategy required at least four threads in total, and the Single-Stage strategy required at least one thread. We also limited the total number of threads to the number of physical cores in our test system. In this setup, each core executes a single "active" thread, which minimizes the impact of thread scheduling on performance. More specifically, the test system has eight cores in total, of which one is always assigned to the PostgreSQL process that implemented the sequential scan, one to the Preprocessor and another one to the Distributor. Hence, at most five threads can be assigned for run-time adaptivity. (The Pipeline Manager was mostly quiescent for these experiments.)

Figure 6 shows the query throughput of the two strategies as we vary the number of Stage threads, using the above methodology. The results show clearly that the Single-Stage strategy consistently outperforms the n-Stage strategy, as long

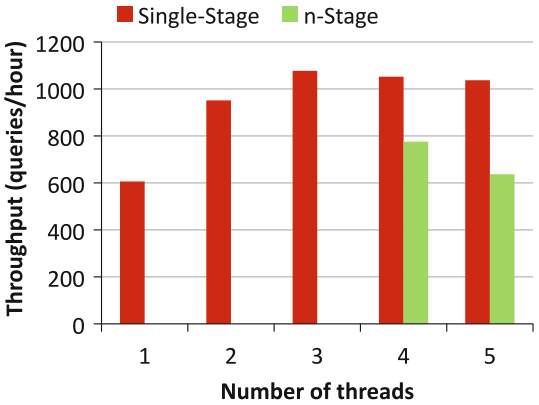

Fig. 6 Performance of adaptivity strategies

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 the cost of L2 data cache misses and thread synchronization, outweighs the benefits gained by the inter-Stage parallelism of the n-Stage strategy. Based on these results, we decide to focus the subsequent experiments on the Single-Stage strategy.

7 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 various workloads. In particular, we focus on the following three high-level questions:

- How does CJOIN throughput scale with increasing numbers of concurrent queries? (Sect. 7.2.1)
- How sensitive is the throughput of CJOIN to workload characteristics? (Sect. 7.2.2)
- How does the size of a data warehouse impact CJOIN 's performance? (Sect. 7.2.3)

7.1 Methodology

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

7.1.1 Systems

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) to ensure that the experimental workloads are executed without obvious performance problems. We 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 so 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 avoid indices and views on the fact table, for the reasons mentioned in previous sections (e.g., to avoid random I/O). For PostgreSQL, we enable the shared-scans feature to maximize its own work sharing; System X does not provide a similar DBA-selectable feature.

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

7.1.2 Data set and workload

We employ the data set and queries defined in the Star Schema Benchmark (SSB) [23]. 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 100 GB 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 substitute in the query template 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 generated by both 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 workload because they contain selection predicates on fact table attributes, and this functionality is not yet supported by our prototype. This modification does not affect the usefulness of the generated

workloads—the omitted queries are the simplest ones in the SSB benchmark and the only ones that do not have a group-by clause.

7.1.3 Evaluation metrics

We quantify the performance 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 start executing 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. Thus, 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). Measuring a fixed set of queries allows us to make meaningful comparisons across different values of n.

7.2 Experimental results

We now present a subset of the experiments we used to evaluate CJOIN .

7.2.1 Influence of concurrency scale

We first evaluate the performance of the three systems as we increase n, the degree of query concurrency. 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, both due to the limited resources and due to the interference among concurrent queries.

Figure 7 shows query throughput for the three systems as a function of n (measurements are gathered with a 100 GB 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 $n = 256$.

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, switching to a more efficient bitmap implementation would likely improve CJOIN scale-up even past 128 concurrent queries.

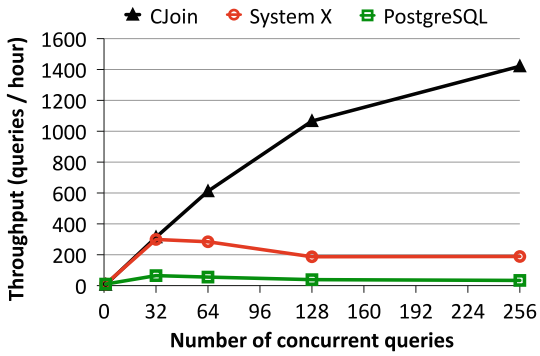

Fig. 7 Query throughput scale-up with number of queries

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 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 8 shows the average response time for queries conforming to template Q4.2 as a function of n. As the number of concurrent queries n increases from 1 to 256, System X's response time grows by a factor of 19× and PostgreSQL's grows by a factor of 66×. These are precisely the undesirable performance patterns that lead to workload fear in existing

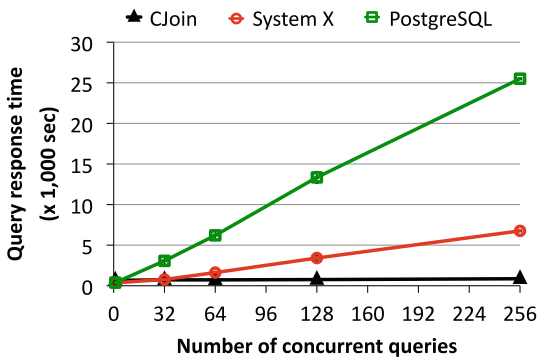

Fig. 8 Predictability of query response time

Table 1 Influence of concurrency on query submission time

# of concurrent queries (n)	32	64	128	256
Submission time (s)	2.4	2.4	2.4	2.3
Response time (s)	724.8	723.1	759.0	861.2

DW platforms: depending on how many *other* queries are running at the same time, one's query completion time could vary by almost two orders of magnitude. CJOIN 's response time, on the other hand, grows by less than 30%—this is excellent, considering that the number of concurrent queries ranges 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.

We now quantify the overhead of query submission in CJOIN as we vary n, focusing again on queries matching template Q4.2. We measure the total time from the submission of the query up until the "start query" control tuple τ_Q is inserted into 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 to CJOIN 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.

7.2.2 *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 proportionally 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 9 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 $s = 10\%$, and we investigate this 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.

CJOIN's performance decreases significantly for higher values of s because the dimension hash tables have to hold an increased number of tuples. This has adverse effects on cache locality and hence access times. Moreover, the overhead of submitting new queries grows substantially, which contributes to the slowdown of the entire pipeline.

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 $s = 10\%$.

7.2.3 *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 (recall that $sf = X$ implies a data set of X GB). Ideally, query throughput is inversely proportional to sf, because queries 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 10 shows normalized query throughput for the three systems as a function of sf (we use a workload of selectivity

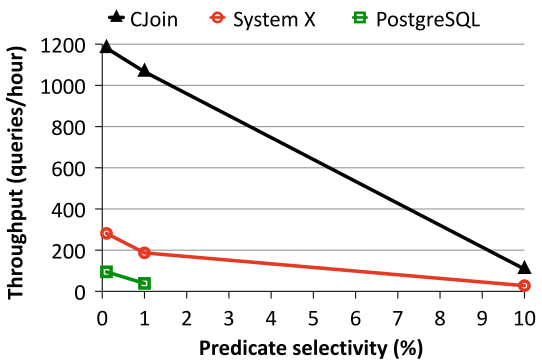

Fig. 9 Influence of query selectivity on throughput

Table 2 Influence of predicate selectivity on query submission time

Predicate selectivity (%)	0.1	1	10
Submission time (s)	1.6	2.4	11.6
Response time (s)	707.2	759.0	3,418.0

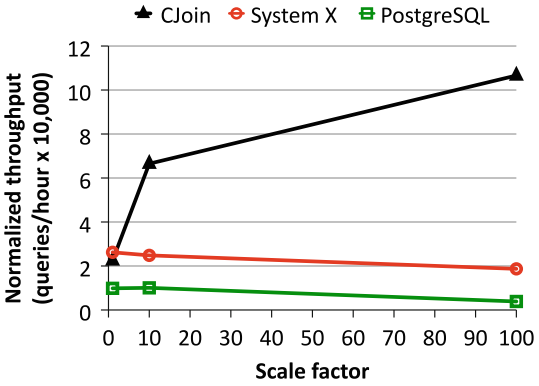

Fig. 10 Influence of data scale on throughput

$s = 1\%$ 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 as sf increases: 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 2× when $sf = 1$ and by a factor of 28× when $sf = 100$.

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 decreasing overhead of new query submission: as sf increases, the overhead relative to query response time decreases (Table 3). 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.

The fact that CJOIN 's normalized throughput increases with the size of the database is an advantage in the context of the rapid increase in data volumes experienced in the data analytics industry today.

7.3 Discussion and outlook

In what follows, we draw conclusions from our empirical results and discuss their relevance to future trends.

7.3.1 Thriving on concurrency

As Fig. 7 indicates, CJOIN 's query throughput increases with the level of query concurrency, in stark contrast to existing DBMSes. As more queries hitch a ride on the bandwagon formed by the streaming of fact tuples through a CJOIN pipeline, the DW system achieves increasingly better utilization of resources. As a result, we expect that, in the future, DBAs will actually *encourage* more users and workloads to run concurrently on the DWs they manage, as opposed to discouraging such practices (which is what happens today). The more queries run at the same time, the more opportunities for amortizing the fixed costs of the infrastructure. At the same time, we expect CJOIN to be a good match for the trend of increasing number of cores per CPU, allowing end-to-end data warehouse performance to better track the evolution of hardware.

7.3.2 Capacity planning

Figure 8 shows that CJOIN provides a substantial improvement in the predictability of query performance: the impact of a DW's existing workload on the performance of a newly submitted query is one to two orders of magnitude lower than in leading DBMSes. This means not only that there is considerably less workload fear, but also that the infrastructure needs for a given expected workload are much easier to estimate and plan. A key property in CJOIN is that the time it takes to execute a query is decoupled from query complexity: each query will take roughly as long as it takes to scan the fact table. This provides a relatively strong form of perfor-

Table 3 Influence of data scale on query submission overhead

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

mance isolation between queries, making query performance mostly a function of data volume. In business environments, capacity planning is a major challenge for IT departments; CJOIN reduces this problem down to estimating the rate of growth in data volume, without requiring complex reasoning about how long it will take to actually analyze that data. Systems whose performance behavior is predictable help reduce the risk of under- or over-provisioning.

7.3.3 Green data analytics

We expect CJOIN to process concurrent joins more energy efficiently than today's DWs. The most important source of energy savings in any DBMS results from reducing the time it takes to answer queries: the less time spent processing queries, the less time the hardware needs to operate at full load, and thus the less energy spent. Second-order savings may also result from the sequential access patterns enabled by CJOIN —the predictability of disk accesses allows the storage subsystem to prefetch and manage the requests more efficiently. In data analytics setups with high query concurrency, the CJOIN architecture allows substantially more value to be derived from the same set of energy-consuming hardware. As the energy footprint of data centers becomes a key bottleneck in IT infrastructure expansion, we expect these opportunities for energy savings to make CJOIN -based data analytics increasingly more attractive.

7.3.4 Memory-resident databases

CJOIN's design 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.

8 From prototype to product

In this section, we present our thoughts on transitioning the design presented here into a data warehouse product serving hundreds of concurrent business analysts in a real-world setting. In particular, we revisit some of the assumptions made in Sect. 2, we discuss ways in which CJOIN can be adapted to accommodate the lifting of these assumptions, and we argue that the CJOIN design is a good choice for the long term. Even though a full treatment of productizing the CJOIN design is beyond the scope of this paper, we provide here an overview of some of the issues and possible solutions.

8.1 Column stores

Column stores have been gaining traction as a scalable system architecture for large data warehouses [1,2,28]. 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 case in which we evaluate filter queries over dimension tables (which occurs as part of a new query registration) is readily handled by the column store, since CJOIN uses the existing query processing infrastructure to retrieve the resulting dimension tuples.

8.2 Compressed tables

Data warehouses may employ compression to reduce the amount of I/O and memory bandwidth used for data transfer [1,25]. CJOIN makes no assumptions about the physical storage of tuples—it only needs to be able 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 fetch 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 [25] to evaluate predicates efficiently on the compressed fact table.

8.3 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 advantage 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. 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, an 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. An optimization is to assign a priority to each partition based on the number of outstanding queries that require it and to access partitions inside each scan in order of priority. This heuristic promotes early query termination and thus increases query throughput.

Note that our CJOIN prototype using *non*-partitioned tables still outperforms PostgreSQL *with* partitioned tables by 287% on SSB [23] with 128 concurrent queries.

Partitioned fact tables constitute one point along a spectrum that offers different performance trade-offs. At one end of this spectrum is an unpartitioned fact table (which results in having to push all fact tuples into the CJOIN pipeline), while at the other end is a fact table with secondary indices on every attribute (which ensures that only the strictly necessary fact tuples are processed). Another point in this spectrum is a design in which CJOIN computes a Bloom filter for each query and has the Preprocessor drop fact tuples that do not match any of the Bloom filters upfront.

8.4 Other DW schemata

Our presentation thus far assumed the common case of using a star schema for data organization, with small dimension tables that can fit in memory. We now discuss the application of CJOIN to three orthogonal extensions of this setting, namely galaxy schemata, schemata with large dimension tables, and schemata that use a snowflake organization of the dimension data.

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 improve 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 feeds 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, we can use CJOIN as a physical operator that efficiently evaluates the "star sub-plans" of bigger query plans.

Large dimension tables may increase significantly the size of the in-memory hash tables and hence the memory requirements of CJOIN . Spilling a large hash table to disk will impact severely the performance of the fact-to-dimension join in the corresponding Filter and hence the processing speed of the whole pipeline. A more efficient alternative is to treat a large dimension table similar to another fact table and hence employ the same mechanism as for galaxy schemata–use CJOIN to evaluate efficiently the star-part of the query on the dominant subset of small dimension tables and then join the results to the large dimension tables in a post-processing phase.

A snowflake schema extends the basic star topology with a hierarchy of normalized dimension tables. Accordingly, a star query may involve additional join predicates between dimension tables on the same hierarchical path. The basic CJOIN mechanism applies virtually unmodified in this setting. The only difference is that each dimension hash table will store tuples that result from the join of several dimension tables.

8.5 Indexes and materialized views

As suggested earlier, fact table indexes are not likely to be useful in the DW setting that we consider, due to the random I/O they induce and potentially high maintenance cost. It is more common (and affordable) for data warehouses to maintain indexes 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 (Algorithm 1).

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. In any case, the query optimizer is free to take advantage of existing materialized views (e.g., aggregation cubes) if a star query can be answered more efficiently this way than with CJOIN .

8.6 Multicore architectures

When turning a prototype into a product, an important question to be answered is what will the longevity of the design be. We believe that CJOIN is a good match for the CPU architectures we can expect in the coming decade. The number of cores per chip is increasing, and the number of CPUs inside a server is increasing as well; the result of this is that servers start looking a lot more like distributed systems [7]. The CJOIN design is well suited for a message-passing environment, because tuples are passed from one stage to the next. The exact structure of the pipeline can be adjusted on the fly, so as to optimize the synchronization/memory operations trade-off. Moreover, CJOIN 's run-time adaptivity strategies can easily take advantage of additional cores by assigning more execution contexts to the Filter pipeline. We therefore see CJOIN well positioned to take advantage of the expected evolution in hardware architectures.

9 Related work

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

9.1 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 [27]. 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.

9.2 Work sharing

Staged database systems [15,16] 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 [11], 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 [31] improve data sharing across concurrent scans by dynamically scheduling queries and their data requests, taking into account current system conditions. Qiao et al. [24] 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. Another possibility is to recycle intermediate results and share them with subsequent queries [17]. Our approach 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 [10,24] 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 [3,8,9,19–21]. 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. Nevertheless, our proposed architecture incorporates elements from continuous query processing, which in turn allows us to transfer techniques from streaming databases to a DW setting. For instance, CJOIN adopts the Grouped Filter operator of Madden et al. [20], but extends it to support fact-to-dimension joins and arbitrary selection predicates; the original operator only supported range predicates on ordered attributes.

9.3 Push-based query processing

CJOIN employs a push-based design, where the fact table is continuously streamed through operators that encode the current workload. A similar principle has been advocated by several recent projects [4,12,30] as the means to achieve high throughput and predictability under high query concurrency.

Crescando [30] is a relational-table implementation that supports a large number of concurrent, single-relation queries. Crescando employs a single continuous scan of the table and forwards each tuple to the set of queries and updates that can consume it. By carefully scheduling the updates and queries on each tuple, Crescando ensures that all operations get a snapshot-isolated view of the data. Similar to CJOIN , all operations have the same predictable performance, since they require a full scan of the table in order to complete.

The Data Cyclotron [12] system employs a similar principle but in a distributed setting. The query processor is formed by several independent peers, linked in a ring topology. Initially, the data are partitioned horizontally across the peers. At run time, each peer forwards data to its successor in the ring and receives data from its predecessor. (The Data Cyclotron uses RDMA to perform network transfers efficiently.) A new query is assigned randomly to a peer for evaluation, and the corresponding plan is executed on the peer for every piece of received data. In a nutshell, data are pushed to the queries on each peer as it flows around the ring. This architecture ensures a load-balanced system with predictable query performance, since queries are assigned randomly to peers

and each query needs to wait for all the data to flow through the peer.

DataPath [4] is a recent proposal (developed independently) that shares several of the principles and ideas behind CJOIN . DataPath performs continuous scans of all the relations and pushes the data through a single execution plan that encodes all concurrent queries. Each plan operator aggregates the computation that is performed by all queries on the same base table or join of base tables. For example, a single operator performs all selections on some input relation R, the same way that the Preprocessor evaluates all predicates on the fact table F. The main idea is to perform all computation relevant for the tuple once it is fetched from the disk to the processor's cache. Similar to CJOIN , each tuple that flows through the plan is augmented with a bitmap that tracks the tuple's relevance to different queries. New queries attach to the plan upon their arrival, and the structure of the plan is optimized on-the-fly based on run-time statistics. Overall, DataPath represents a new system architecture that employs several of the principles behind CJOIN in order to achieve predictable query performance. CJOIN has the same goal but with a different scope: it is easier to integrate in an existing database system, and it is highly specialized for the common class of star queries.

9.4 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 [14,26]. 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.

9.5 Constant time query processing

BLINK [25] 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 de-normalized, and our design directly supports high query concurrency, whereas BLINK targets the execution of one query at a time.

10 Conclusion

We presented the design of CJOIN , a novel operator for the concurrent evaluation of tens to hundreds of star-schema queries. CJOIN shares across multiple queries the common parts of the queries' execution plans, without requiring the queries to be optimized or even submitted in a batch. We presented an empirical evaluation of CJOIN using the Star Schema Benchmark. Our results demonstrate that CJOIN consistently outperforms both a widely used commercial RDBMS and PostgreSQL on a variety of workloads. CJOIN delivers one to two orders of magnitude improvement in both query response time and predictability of this response time when executing 256 concurrent queries. As the database size increases, the normalized query throughput in CJOIN increases as well, compared to the other evaluated systems, in which it decreases. We conclude that CJOIN provides a promising design for achieving predictable performance and high query concurrency in data analytics systems.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2006)
2. Abadi, D.J., Myers, D.S., Dewitt, D.J., Madden, S.R.: Materialization strategies in a column-oriented DBMS. In: Proceedings of International Confernce on Data Engineering (2007)
3. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of International Conference on Data Engineering (2008)
4. Arumugam, S., Dobra, A., Jermaine, C.M., Pansare, N., Perez, L.: The DataPath system: a data-centric analytic processing engine for large data warehouses. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2010)
5. Aster Data Systems: Personal communication between George Candea and multiple Aster Data customers (2009)
6. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2004)
7. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The Multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of ACM Symposium on Operating Systems Principles (2009)
8. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2003)
9. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for internet databases. SIGMOD Record **29**(2), 379–390 (2000)
10. Cieslewicz, J., Ross Kenneth, A.: Adaptive aggregation on chip multiprocessors. In: Proceedings of International Conference on Very Large Data Bases (2007)

11. Fernandez, P.M.: Red Brick warehouse: A read-mostly RDBMS for open SMP platforms. In: Proceedings of ACM SIGMOD International Conference on Management of Data (1994)
12. Goncalves, R., Kersten, M.: The data cyclotron query processing scheme. In: Proceedings of ACM International Conference on Extending Database Technology (2010)
13. Hanani, M.Z.: An optimal evaluation of boolean expressions in an online query system. Commun. ACM **20**(5), 344–347 (1977)
14. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: Proceedings of ACM SIGMOD International Conference on Management of Data (1996)
15. Harizopoulos, S., Ailamaki, A.: StagedDB: Designing database servers for modern hardware. IEEE Data Eng. Bull. **28**(2), 11–16 (2005)
16. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: Qpipe: a simultaneously pipelined relational query engine. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2005)
17. Ivanova, M.G., Kersten, M.L., Nes, N.J., Goncalves, R.A.: An architecture for recycling intermediates in a Column-store. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2009)
18. Liu, Z., Parthasarathy, S., Ranganathan, A., Yang, H.: A generic flow algorithm for shared filter ordering problems. In: Proceedings of Symposium on Principles of Database Systems. New York (2008)
19. Liu, Z., Parthasarathy, S., Ranganathan, A., Yang, H.: Near-optimal algorithms for shared filter evaluation in data stream systems. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2008)
20. Madden, S., Shah, M., Hellerstein Joseph, M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of ACM SIGMOD International Conference on Management of Data (2002)
21. Majumder, A., Rastogi, R., Vanama, S.: Scalable regular expression matching on data streams. In: Proceedings of International Conference on Data Engineering (2008)
22. Munagala, K., Babu, S., Motwani, R., Widom, J.: The pipelined set cover problem. In: Proceedings of International Conference on Database Theory (2005)
23. Patrick O'Neil, E.B.O., Chen, X.: The Star Schema Benchmark. (2007). http://www.cs.umb.edu/~poneil/StarSchemaB.PDF
24. Qiao, L., Raman, V., Reiss, F., Haas, P., Lohman, G.: Main-memory scan sharing for multi-core CPUs. In: Proceedings of International Conference on Very Large Data Bases (2008)
25. Raman, V., Swart, G., Qiao, L., Reiss, F., Dialani, V., Kossmann, D., Narang, I., Sidle, R.: Constant-time query processing. In: Proceedings of International Conference on Data Engineering (2008)
26. Roussopoulos, N., Chen, C.M., Kelley, S., Delis, A., Papakonstantinou, Y.: The ADMS project: views R Us. IEEE Data Eng. Bull. **18**(2) (1995)
27. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
28. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: Proceedings of International Conference on Very Large Data Bases (2007)
29. TPC benchmark DS (decision support), draft specification, revision 32. http://www.tpc.org/tpcds/spec/tpcds32.pdf
30. Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., Kossmann, D.: Predictable performance for unpredictable workloads. Proc. VLDB Endow. **2**(1), 706–717 (2009)
31. Zukowski, M., Héman, S., Nes, N., Boncz, P.: Cooperative scans: dynamic bandwidth sharing in a DBMS. In: Proceedings of International Conference on Very Large Data Bases (2007)