Optimization of Continuous Queries in Federated Database and Stream Processing Systems

Yuanzhen Ji[1], Zbigniew Jerzak[1], Anisoara Nica[2], Gergor Hackenbroich[1], Christof Fetzer[3]

[1]SAP SE Dresden Germany, [2]SAP SE Waterloo Canada
[3]Systems Engineering Group, TU Dresden, Germany
[1,2]{firstname.lastname}@sap.com, [3]christof.fetzer@tu-dresden.de

Abstract: The constantly increasing number of connected devices and sensors results in increasing volume and velocity of sensor-based streaming data. Traditional approaches for processing high velocity sensor data rely on stream processing engines. However, the increasing complexity of continuous queries executed on top of high velocity data has resulted in growing demand for federated systems composed of data stream processing engines and database engines. One of major challenges for such systems is to devise the optimal query execution plan to maximize the throughput of continuous queries.

In this paper we present a general framework for federated database and stream processing systems, and introduce the design and implementation of a cost-based optimizer for optimizing relational continuous queries in such systems. Our optimizer uses characteristics of continuous queries and source data streams to devise an optimal placement for each operator of a continuous query. This fine level of optimization, combined with the estimation of the feasibility of query plans, allows our optimizer to devise query plans which result in 8 times higher throughput as compared to the baseline approach which uses only stream processing engines. Moreover, our experimental results showed that even for simple queries, a hybrid execution plan can result in 4 times and 1.6 times higher throughput than a pure stream processing engine plan and a pure database engine plan, respectively.

1 Introduction

The increasing amount of connected devices and sensors has led to a surge in the amount, velocity, and value of streaming sensor data. The increasing value of information carried by the streaming sensor data motivates the need to combine and query such data streams. Traditional approaches for processing streaming sensor data rely on stream processing engines (SPE) using continuous queries. A continuous query is issued once and is executed constantly over the data streams, returning a continuous stream of query results.

Existing SPEs are built either from scratch ([ACc+03, KS04, Esp, Pro], etc.) or on top of existing database systems [CH10, FKC+09, LGI09]. Despite this fact, they show limitations in processing certain types of complex continuous queries when compared to modern databases, such as in-memory column stores [Ji13]. In addition, before introducing SPEs, most of today's enterprises already have database systems in place for data persistence and on-demand analytical processing. Hence, the co-existence of a SPE and a DBMS can be found in many real-world setups. Following the philosophy that "no one size fits all", and aiming to explore the potential of such SPE-database setup, in this paper, we propose to federate the SPE and the database engine for joint execution of continuous queries to achieve performance which cannot be matched by either engine alone. By “federate”, we mainly mean outsourcing certain fragments of a continuous query from the SPE to the database engine when the outsourcing can lead to better performance; the federated system, however, supports queries that access both streaming data and stored data as well.

One major challenge of such systems is to find the optimal execution plan for a given continuous query. Existing federated database and stream processing systems either have no federated optimizer at all [BCD$^+$10], or choose the most suitable system for the entire query [LHB13]. Moreover, none of them have considered the *feasibility* property of execution plans of continuous queries, which describes the capability of a plan to keep up with the data arrival rate [AN04]. Finally, the heterogeneity between the underlying SPE and the database engine causes the *non-additivity* of the query execution cost [DH02]. Specifically, the non-additive execution cost means that the cost of executing two consecutive operators in the database engine is not necessarily higher than the cost of executing only the first operator in the database engine. This non-additivity makes it difficult for a query optimizer to make pruning decisions during plan enumeration. Existing solutions used in traditional database systems for handling non-additivity must be extended to consider the feasibility property of plans of continuous queries.

Our major contributions in this paper is the design and implementation of a static cost-based optimizer for optimizing relational continuous queries in a federated database and stream processing system. Our optimizer fully exploits the potential of distributed execution of continuous queries across a SPE and a database engine. Using characteristics of queries and data streams, our optimizer determines an optimal placement for each operator in a continuous query, taking into account the feasibility of query plans and the non-additivity of the query execution cost caused by the federation. To reduce the search space of query plans, we adopt the two-phase optimization strategy [HS91], which is widely used in federated or parallel database systems, as well as systems with heterogeneous multicore architectures (e.g., [HLY$^+$09]). In Phase-One, an optimal logical query plan is produced; in Phase-Two, placement decisions for all operators in the chosen logical plan are made. We further exploit plan-pruning opportunity in Phase-Two based on the study of the cost characteristics of operators placed on the two different engines, thereby reducing the search space further.

As a proof of concept, we federate a commercial stream processing engine—SAP ESP [Pro], and a columnar in-memory database system (IMDB)—SAP HANA [SFGL13], for joint execution of continuous queries. We have implemented the proposed optimization approach by directly extending the optimizer of the IMDB. We experimentally demonstrate that our fine level of optimization, combined with the estimation of the feasibility of query plans, can devise query plans which result in up to 8 times higher throughput when compared to the baseline—the pure SPE-based execution. Our experimental results showed that even for simple queries, the optimizer can derive non-obvious decisions which result in up to 4 times higher throughput when compared to the pure SPE-based execution and up to 1.6 times higher throughput when compared to the pure IMDB-based execution. Our experimental results confirm the superiority and the necessity of a federated optimizer for continuous queries working at the operator level.

Note that as pointed in [AN04], static query optimization is a valid approach when the characteristics of input streams change slowly or the pattern of change is predictable, which is often observed in data streams originating from sensors with fixed reporting frequencies. Before moving on to a dynamic optimization solution, we must first understand what can be achieved by doing static optimization for continuous queries in a federated database and stream processing system, which is the goal of this paper.

The remainder of this paper is organized as follows. Following the background introduction in Section 2, Section 3 gives an overview of continuous query execution in our prototype federated database and stream processing system. Section 4 defines the query optimization objective in such federated systems. Section 5 drills down to the cost model adopted in our optimizer, followed by descriptions of our two-phase optimization approach in Section 6. Section 7 presents the pruning strategy applied in the second phase of the optimization. In section 8 we experimentally study the effectiveness of our optimizer. We discuss related work in Section 9 and conclude in Section 10.

2 Background

This section presents the semantics of continuous queries adopted in our work (Section 2.1), as well as basics about the pipelined execution model (Section 2.2).

2.1 Continuous Query Semantics

Although a few studies, such as [ABW06, KS09, BDD$^+$10], have tried to offer clean semantic models for continuous queries executed over data streams, to date, there is no established standards. In our work, we adopt the abstract semantics defined in [ABW06], which is based on two data types, *stream*s and *time-varying relation*s, and three classes of query operators. Assuming a discrete and ordered time domain $\mathcal{T}$, streams and time-varying relations are defined as follows:

Stream A stream S is a possibly infinite bag of elements $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the timestamp of s.

Time-varying Relation A time-varying relation R is a mapping from $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to the schema of R. In the following, we call time-varying relations as relations for short wherever the context of stream processing is clear.

The three classes of query operators are the *stream-to-relation* (S2R) operators, which produce one relation from one stream; *relation-to-relation* (R2R) operators, which produce one relation from one or more relations; and *relation-to-stream* (R2S) operators, which produce one stream from a relation. The most typical S2R operator is the *window* operator. There are various types of window operators [ABW06, PS06]. In this paper, we focus on time-based and tuple-based sliding windows. R2R operators are straightforward counterparts of relational operators in conventional database systems. We focus on selection, projection, equi-join, and aggregation in this paper. Without loss of generality, we assume that each R2R operator has at most two input relations; a multi-way join is treated as a sequence of two-way joins. We adopt the following semantics for sliding-window aggregations: aggregation results are produced at each slide of the window.

Table 1: Notations for representing logical and physical plans of continuous queries.

Notation	Description
$T = (\mathcal{O}, \mathcal{E})$	A logical plan of a continuous query CQ
$O_i \in \mathcal{O}$	A logical operator in T
$e_{ij} \in \mathcal{E}$	Data flow from O_i to O_j
$\mathcal{P}(T) = (\mathcal{O}', \mathcal{E}', \mathcal{M})$	A physical plan of T
$O^{spe} \in \mathcal{O}'$	A basic physical operator running in the SPE
$O^{db} \in \mathcal{O}'$	A migration candidate (composite operator) running in the database
$e'_{ij} \in \mathcal{E}'$	Data flow from O_i^x to O_j^y, $x, y \in \{spe, db\}$
$\mathcal{M}(O^x)$	The set of logical operators in T that O^x maps to, $x \in \{spe, db\}$

We have decided to adopt the above described continuous query semantics, because time-varying relations and R2R operators have straightforward semantic mapping to conventional relations and query operators in database systems, respectively. Hence, it provides a sound semantics foundation for federated execution of continuous queries.

2.2 Pipelined Execution

We consider the pipelined query execution model [Gra93], which is adopted by most existing SPEs (e.g., STREAM [ABW06], Aurora [ACc$^+$03]) to adapt to the "push" characteristic of data streams. With pipelined execution, query operators are organized into series of producer-consumer pairs that are connected via a buffering mechanism, e.g., data queues. The producer and the consumer can run in parallel, embodying the so-called *pipelined parallelism* [HM94]. Pipelined execution allows exploiting the power of modern multiprocessor machines to accelerate data processing.

We model pipelined relationships among operators in a continuous query CQ with a directed tree, denoted as $T = (\mathcal{O}, \mathcal{E})$. A node $O_i \in \mathcal{O}$ represents a query operator and an edge $e_{ij} \in \mathcal{E}$ represents the data flow from node O_i to O_j. Similar to [ABW06, KS09], we adopt notions used in conventional database systems, and refer to such a tree as a *logical plan* of CQ. Operators in a logical plan are referred to as logical operators. A logical plan of a continuous query may have multiple semantically equivalent alternatives. Notations used for representing continuous queries are summarized in Table 1.

3 Federated Continuous Query Execution

We have built a prototype system which consists of a state-of-the-art SPE (SAP ESP) and a columnar IMDB (SAP HANA). In this section, we give an overview of continuous query execution in our system.

Determined by the semantic mapping between continuous queries and SQL queries, given a logical plan T of a query, fragments of T that can potentially be executed in the database engine are sub-trees of T that contain only R2R operators. We call such a sub-tree of T as a *migration candidate*. A composition of several R2R operators produces one relation from one or more relations, and hence can be regarded as a R2R operator as well (see Section 2.1). We regard each migration candidate as a *composite R2R operator*. A migration candidate can be translated into a SQL query and executed in the database engine. Particularly, base relations involved in the SQL query map to the input relations of the migration candidate; the result of the SQL query maps to the output relation of the migration candidate.

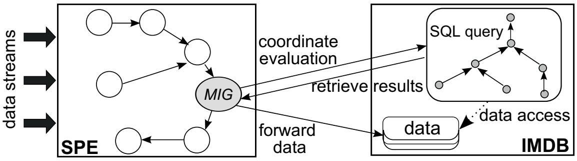


Figure 1: Execution of continuous queries across a SPE and an IMDB.

Figure 1 illustrates how a continuous query is executed across the SPE and the IMDB in our system. The SPE acts as the gateway of external data streams. Federated query execution involves data transfer between the SPE and the database engine. Specifically, for each migration candidate placed in the database engine, we need to transfer relevant input data from the SPE to the database engine; and transfer execution results from the database engine back to the SPE.

To retain the original query semantics, the SQL query corresponding to a migration candidate must be re-executed in response to changes in the input relations of the migration candidate. To coordinate the data transfer between the two engines and the re-execution of the corresponding SQL query, we introduce a new operator *MIG* into the SPE. A *MIG* operator acts as a wrapper of a migration candidate executed in the IMDB. It controls the data transfer between the two engines and hides the execution specifics within the IMDB from the SPE. In a parallel environment, *MIG* operators run in parallel with other query operators in the SPE. However, from the SPE's perspective, each migration candidate wrapped by a *MIG* operator is a black-box, and the original pipelined relationships among operators in the migration candidate is no longer visible.

Execution Plan Representation. Given a logical plan $T = (\mathcal{O}, \mathcal{E})$ of a continuous query, we denote an execution plan of T as $\mathcal{P}(T) = (\mathcal{O}', \mathcal{E}', \mathcal{M})$. $O_i^x \in \mathcal{O}'$ represents a physical operator in the execution plan, where $x \in \{spe, db\}$. Specifically, O_i^{spe} represents a basic query operator (selection, join, etc.) placed in the SPE, and O_i^{db} represents a migration candidate placed in the database engine. For ease of reference, in the remainder of this paper, we refer to a basic query operator placed in the SPE as a *SPE-op* and a composite operator representing a migration candidate placed in the database engine as a *DB-op*. $e'_{ij} \in \mathcal{E}'$ represents the data flow from O_i^x to O_j^y ($x, y \in \{spe, db\}$). Finally, $\mathcal{M}$ defines a mapping from $\mathcal{O}$ to $\mathcal{O}'$. For each $O^x \in \mathcal{O}'$, $\mathcal{M}(O^x)$ defines the subset of $\mathcal{O}$ that O^x maps to. Specifically, $\mathcal{M}(O^{spe})$ is a set containing only one logical operator; $\mathcal{M}(O^{db})$ is a set containing one or more logical operators. Notations used for representing physical plans are summarized in Table 1 as well.

4 The Optimization Objective

A common performance metric for a continuous query executed over data streams is the output rate of the query [GO03]. Therefore, maximizing the query output rate is a widely adopted objective in continuous query optimization [AN04, VN02]. Maximizing the output rate of a query is equivalent to maximizing the amount of input data processed by the query in unit time, which we define as the *query throughput* in this paper. Intuitively, an execution plan reaches its maximum throughput when it can keep up with the data arrival rate. This capability of keeping up with the data arrival rate is defined as the *feasibility* of the plan [AN04]. A continuous query is a *feasible query* if it has at least one feasible plan.

The optimization objective on the query throughput suggests that a query optimizer should favor feasible plans over infeasible plans for feasible queries, and should pick the plan that can maximize the query throughput for infeasible queries. However, what if a query has multiple feasible plans? It has been shown in [AN04] that given enough resources, all feasible plans of a continuous query have the same throughput. Therefore, in this case, we apply a different optimization objective—that is, minimizing the total resource utilization of the query. The motivation behind is that intuitively, the less resources each query consumes, the more number of queries that a system can execute concurrently. In summary, our optimization objective is as follows:

- For feasible queries, find the feasible execution plan which has the least resource utilization.
- For infeasible queries, find the plan which has the maximum the query throughput.

Generally, given two execution plans of a continuous query, possible situations faced by an optimizer, and the respective appropriate optimization decision are the following:

- Situation 1: One plan is feasible and the other is infeasible. → Choose the feasible plan.
- Situation 2: Both plans are feasible. → Choose the one with less resource utilization.
- Situation 3: Both plans are infeasible. → Choose the one with higher throughput.

Discussion. Ayad et al. [AN04] adjust the above optimization objectives to incorporate the influence of load shedding. They insert load shedding operators into plans of an infeasible query, thereby turning all infeasible plans into feasible ones. In this paper, we focus on discussing continuous query optimization in a federated environment and do not consider applying load shedding for infeasible queries.

5 The Cost Model

To achieve the optimization objective described in the previous section, we propose a cost-based optimizer. Without loss of generality, we consider continuous queries whose logical plans have window operators appear only as leaf nodes and R2S operators appear only as root nodes. Note that a query with window or R2S operators appearing as internal nodes can always be split into a set of sub-queries, with the logical plan of each sub-query satisfying the above condition. We also assume a highly parallel environment with abundant memory for query execution. Hence, operators are fully pipelined and do not time-share CPU resources.

We assume that data from source streams arrive at a relatively stable rate. The data rate λ_{S_i} of each source stream S_i in a plan $\mathcal{P}$ defines how much data from S_i should be processed by $\mathcal{P}$ within unit time. We refer to data arrived from all source streams in a plan within unit time as the *unit-time source arrivals*. We further define the amount of data that an operator produces as a result of the unit-time source arrivals as the *source-driven output-size* of the operator, denoted by λ_{out}. Note that (1) the source-driven output-size of an operator is the amount of data produced by the operator as a result of unit-time source-arrivals, but is not the amount of data generated by the operator within unit time, which is also know as the output rate; (2) the source-driven output-size of an operator O_i is used as the *source-driven input-size* (denoted by λ_{in}) by its direct downstream operator O_j to estimate the source-driven output-size of O_j.

Given data rates of all source streams involved in a query, the source-driven output-size λ_{out} of each operator can be estimated in a bottom-up way. In this paper, we adapt the method proposed in [AN04] under our query semantics model (see Section 2.1) to estimate λ_{out} of window-based selection, projection, and join. Specifically, for a selection or a projection with selectivity f[1], its source-driven output-size is

$$\lambda_{out} = f\lambda_{in}. \tag{1}$$

For a join operator, suppose that the size of its left input relation is W_L, the size of the right input relation is W_R, and the selectivities relative to the left and the right relations are f_L and f_R, respectively. Its source-driven output-size can be estimated by Eq. (2). The size of a relation is defined as the number of tuples contained in the relation, which can be estimated in a bottom-up way as described in [AN04].

$$\lambda_{out} = \lambda_{in_L} f_R W_R + \lambda_{in_R} f_L W_L \tag{2}$$

Recall that we define aggregate operators to produce results at each slide of the upstream window (see Section 2.1). For a time-based sliding window, if the slide size is β time units, then on average the unit-time sliding frequency, denoted as l, is $1/\beta$. For a tuple-based sliding window whose slide size is β tuples, the sliding frequency depends on the data rate of the source stream as well, and is estimated as $l = \lambda_s/\beta$. Suppose that the average number of result groups, as determined by the associated grouping predicate of the aggregate operator, is g. We estimate the source-driven output-size of an aggregate operator as

$$\lambda_{out} = lg. \tag{3}$$

5.1 Operator Cost

Having introduced the estimation of source-driven input/output-sizes of operators, we are ready to estimate costs of physical operators in an execution plan. Each tuple arriving at an operator requires some processing effort from the operator. We define the average time that an operator O_j^x requires to process a single tuple from a direct upstream operator O_i^x as the *unit processing cost of O_j^x for O_i^x*, denoted by c_{ji}, or simply c_j if O_j^x has only one direct upstream operator. For an operator O_j^x with k upstream operators, we define the total cost of O_j^x caused by unit-time source-arrivals as the *source-driven input processing cost*, and denote it by u_j. We estimate u_j as

$$u_j = \sum_{i=1}^{k} \lambda_i c_{ji}. \tag{4}$$

To keep up with the data arrival rate, the time needed to process a single tuple by each operator in a pipeline must be shorter than the average data arrival interval at the operator. In other words, the constraint $\sum_{i=1}^{k} \lambda_i c_{ji} \leq 1$, namely $u_j \leq 1$, must hold [AN04, VN02]. An operator that cannot meet the this constraint is the *bottleneck* of the pipeline.

Cost of *SPE-ops*. The cost estimation method described above can be used directly to estimate costs of *SPE-op*s in an execution plan. The unit processing cost c of a specific *SPE-op* depends on the type and the physical implementation of the operator.

[1] The selectivity of a projection is 1.

Cost of *DB-op*s. In contrast to a *SPE-op*, which maps to a single logical operator, a *DB-op* maps to one or more logical operators and is evaluated as one single SQL query. Hence, the unit processing cost of a *DB-op* is practically the execution cost of the corresponding SQL query. Moreover, each time when a *DB-op* is executed, we need to transfer the relevant input data from the SPE, and the execution results back to the SPE (see Section 3). The costs of inter-engine data transfer must be taken into account as well. In summary, the unit processing cost of a *DB-op* consists of three parts: the cost of transferring relevant input data from the SPE to the database, the cost of evaluating the SQL query, and the cost of transferring the SQL query results back to the SPE. In our prototype system, we extended and tuned the built-in cost model of the IMDB to estimate the cost of *DB-op*s.

5.2 Execution Plan Cost

Based on the cost estimation for individual operators described in Section 5.1, we now introduce the cost model for a complete execution plan.

Corresponding to the optimization objectives defined in Section 4, we define the cost of an execution plan $\mathcal{P}$ with m operators, denoted by $C_u(\mathcal{P})$, as a two dimensional vector consisting of two cost metrics: the *bottleneck cost* $C_b(\mathcal{P})$ and the *total utilization cost* $C_u(\mathcal{P})$; namely, $C(\mathcal{P}) = \langle C_b(\mathcal{P}), C_u(\mathcal{P})\rangle$. $C_b(\mathcal{P})$ and $C_u(\mathcal{P})$ are computed as follows:

$$C_b(\mathcal{P}) = max\{u_j^x : j \in [1, m]\}. \tag{5}$$

$$C_u(\mathcal{P}) = \sum_{j=1}^{m} u_j^x \tag{6}$$

Note that here the "bottleneck" refers to the operator with the highest source-driven input processing cost in the plan. We use the bottleneck cost to check the feasibility of a plan. Moreover, for infeasible plans of a query, a higher bottleneck cost implies that the plan can handle fewer input data per unit time; therefore, we also use the bottleneck cost as an indicator of the throughput of an infeasible plan. The total utilization cost estimates the total amount of resources required by the plan to process unit-time source arrivals.

Based on the above cost metrics for execution plans, we define the *optimal plan* of a given continuous query as follows:

Definition 1. *For a continuous query CQ, an execution plan $\mathcal{P}$ is an optimal plan of CQ, iff for any other plan $\mathcal{P}'$ of CQ, one of the following conditions is satisfied:*

Condition 1°: $C_b(\mathcal{P}) \leq 1 < C_b(\mathcal{P}')$
Condition 2°: $C_b(\mathcal{P}) \leq 1$, $C_b(\mathcal{P}') \leq 1$, and $C_u(\mathcal{P}) \leq C_u(\mathcal{P}')$
Condition 3°: $1 < C_b(\mathcal{P}) \leq C_b(\mathcal{P}')$

Each condition in Definition 1 applies in a specific situation described in Section 4. Condition 1° is applied when $\mathcal{P}$ is feasible and $\mathcal{P}'$ is infeasible; Condition 2° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are feasible; and Condition 3° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are infeasible.

6 Two-Phase Optimization

In principle, a R2R operator of a query can be executed either in the SPE or in the database engine. However, the placement decision for the operator does not influence its pipelined relationships with its upstream and downstream operators. Consequently, the options of the execution engine for an operator can be treated as physical implementation alternatives of the operator [BCE$^+$05], thereby allows integrating the selection of the execution engine for operators into the physical plan enumeration phase of a query optimizer.

A continuous query could have a large number of semantically equivalent logical plans due to, for instance, different join ordering possibilities. Even for an individual logical plan T with n R2R operators, there are in total 2^n possible execution plans for T. Due to the large search space of execution plans, exhaustive search for the optimal plan is too expensive. In this paper, following the idea applied in many existing federated, distributed, or parallel database systems, we adopt a *two-phase* optimization approach [HS91]. Specifically, the optimization process is divided into Phase-One, which determines the optimal logical plan for a given query, considering the join ordering and the push-down/up of aggregates, etc.; and Phase-Two, which determines the execution engines of operators in the logical plan picked in Phase-One.

The System R style dynamic programming optimizer [SAC$^+$79] is a widely used query optimizer in existing database systems. It relies on the so-called *principle of optimality* to prune away expensive plans as early as possible. We would like to adopt the System R style optimization approach in our optimizer as well, to find the optimal logical plan in Phase-One. However, to be able to use this approach, we must first show that the principle of optimality holds in the context of continuous query optimization as well; namely, the optimal plan for joining a set of k streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ with another stream S_{k+1} can be obtained by joining stream S_{k+1} with the optimal plan that joins all streams in $\mathcal{S}$.

Let us consider the join query in Figure 2. The window operators are skipped for brevity. We denote the optimal plan for joining the set of streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ as $\mathcal{P}_{opt}$. Any suboptimal plan is denoted as $\mathcal{P}_s$. Suppose that the next stream to be joined is S_{k+1}, which incurs λ_R unit-time source-driven arrivals at the new join operator (denoted as $\bowtie_{k+1}$). Note that the total number of join results produced by $\mathcal{P}_{opt}$ as a result of unit-time arrivals from all streams in $\mathcal{S}$ is the same as that produced by $\mathcal{P}_s$. Namely, the source-driven output-sizes of $\bowtie_k$ are identical in all plans that join streams in $\mathcal{S}$. Hence, according to Eq. (4), we can infer that the source-driven input processing cost u of $\bowtie_{k+1}$ is the same in all plans extended from plans for $\bowtie_k$. Denoting the plan extended from $\mathcal{P}_{opt}$ to join with S_{k+1} as $\mathcal{P}'_{opt}$, and the plan extended from $\mathcal{P}_s$ to join with S_{k+1} as $\mathcal{P}'_s$, we now prove that $\mathcal{P}'_{opt}$ is still optimal compared to any $\mathcal{P}'_s$.

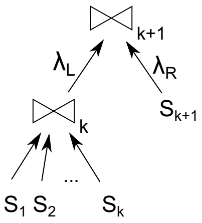


Figure 2: Illustrative logical plan that extends the subplan joining a set of streams $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ to join with another stream S_{k+1}.

Proof Sketch.

Case 1: $\mathcal{P}_{opt}$ is feasible. In this case, a plan $\mathcal{P}_s$ is suboptimal either because it is infeasible (Condition 1° in Definition 1), or because it is feasible as well but has a higher total utilization cost (Condition 2°).

- **Case 1.1:** If $\mathcal{P}_s$ is infeasible, then the plan $\mathcal{P}'_s$ extended from $\mathcal{P}_s$ with $\bowtie_{k+1}$ is still infeasible. Extending $\mathcal{P}_{opt}$ with $\bowtie_{k+1}$ can either leave the resulting plan $\mathcal{P}'_{opt}$ feasible if $u \leq 1$, or make $\mathcal{P}'_{opt}$ infeasible if $u > 1$. In the former case, it is obvious that $\mathcal{P}'_{opt}$ is better than $\mathcal{P}'_s$. In the later case, we must compare the bottleneck costs of $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$. $C_b(\mathcal{P}'_{opt})$ now equals u. $C_b(\mathcal{P}'_s)$ equals $C_b(\mathcal{P}_s)$ if $u < C_b(\mathcal{P}_s)$, or u if $u \geq C_b(\mathcal{P}_s)$. In either case, we have $1 \leq C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$. Therefore, $\mathcal{P}'_{opt}$ is still optimal (Condition 3°).

- **Case 1.2:** If $\mathcal{P}_s$ is also feasible but has a higher total utilization cost than $\mathcal{P}_{opt}$, then the feasibility of $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ is determined by u in the same way. Specifically, if $u \leq 1$, then both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are feasible. Moreover, $C_u(\mathcal{P}'_s)$ is higher than $C_u(\mathcal{P}'_{opt})$, because $C_u(\mathcal{P}'_s) = C_u(\mathcal{P}_s)+u$, $C_u(\mathcal{P}'_{opt}) = C_u(\mathcal{P}_{opt})+u$, and $C_u(\mathcal{P}'_s) > C_u(\mathcal{P}'_{opt})$. Therefore, $\mathcal{P}'_{opt}$ is optimal compared to $\mathcal{P}'_s$ according to Condition 1°. If $u > 1$, then both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are infeasible, and we have $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u > 1$. Therefore, $\mathcal{P}'_{opt}$ is still optimal according to Condition 3°.

Case 2: $\mathcal{P}_{opt}$ is infeasible. In this case, $\mathcal{P}_s$ can be suboptimal only when $\mathcal{P}_s$ is infeasible and $1 < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$ (Condition 3°). Plans extended from infeasible plans remain infeasible. Therefore, both $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are infeasible. Depending on the value of u, the relationship between $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$ is one of the following cases:

- If $u < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$, then $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}_{opt}) \leq C_b(\mathcal{P}'_s) = C_b(\mathcal{P}_s)$.
- If $C_b(\mathcal{P}_{opt}) \leq u < C_b(\mathcal{P}_s)$, then $C_b(\mathcal{P}'_{opt}) = u < C_b(\mathcal{P}'_s) = C_b(\mathcal{P}_s)$.
- If $C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s) \leq u$, then $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u$.

We can observe that $1 < C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$ (Condition 3°) holds in all three cases. Hence, $\mathcal{P}'_{opt}$ is still optimal. □

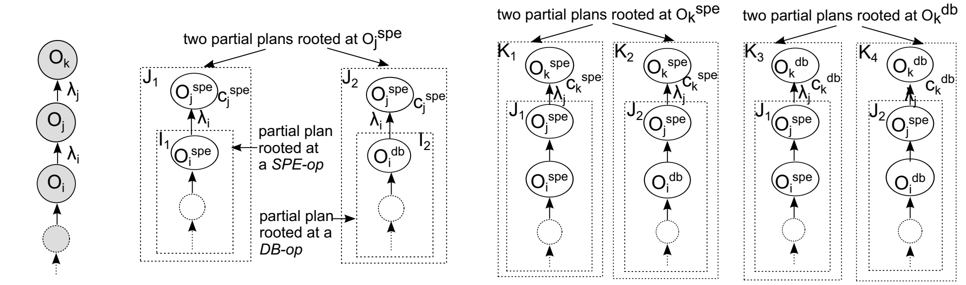
Discussion. The above proof shows that the key reasons for the applicability of the principle of optimality are: (1) the source-driven input processing cost u of the new join operator $\bowtie_{k+1}$ is the same in all plans extended from a possible plan that joins streams $S_1, S_2, \ldots, S_k$; (2) u of $\bowtie_{k+1}$ does not change when extending $\bowtie_{k+1}$ to join with other streams.

7 Pruning in Phase-Two

Taking the plan produced in Phase-One as an optimal logical plan, our optimizer determines in Phase-Two the execution engine for each operator in the plan in a bottom-up way. In this section, we describe the pruning strategy used by our optimizer in Phase-Two to further reduce the search space and prove its validity.

By studying the characteristics of the cost of individual *SPE-op*s and *DB-op*s, as well as the influence of their costs on the cost of the entire execution plan, we have observed the following properties of *SPE-op*s: (1) the source-driven input processing cost u of a *SPE-op* O^{spe} is identical in all partial plans rooted at O^{spe}; (2) the source-driven input processing cost of O^{spe} in a partial plan $\mathcal{P}$ rooted at O^{spe} is not changed when $\mathcal{P}$ is further extended. In fact, these two properties are similar to that of the join operators in Figure 2, which suggests that we can apply a similar principle of optimality for pruning. Specifially, to obtain an optimal (partial) plan rooted at a *SPE-op* O^{spe}, it suffices to consider only the optimal partial plans rooted at the direct upstream operators of O^{spe}.

Let us consider the logical plan shown in Figure 3a. Suppose that the current logical operator being enumerated is O_j. Because we adopt a bottom-up enumeration approach,



(a) logical plan

(b) Two *SPE-op* rooted partial plans until O_j, which are constructed from two partial plans until O_i

(c) Two *SPE-op* rooted partial plans and two *DB-op* rooted partial plans until O_k, which are constructed based on the two partial plans until O_j in (b).

Figure 3: Pruning opportunities when enumerating partial plans rooted at a *SPE-op*.

the enumeration for O_i should have completed. Also suppose that we obtain in total two partial plans until O_i, denoted by I_1 and I_2 (see Figure 3b). I_1 is rooted at a *SPE-op* and I_2 is rooted at a *DB-op*. If we do not consider pruning, we can construct two *SPE-op* rooted partial plans until O_j; one plan extends I_1, denoted by J_1, and the other plan extends I_2, denoted by J_2. We now prove that indeed we need to construct only one *SPE-op* rooted partial plan until O_j, based on the optimal partial plan between I_1 and I_2.

Proof Sketch. This proof consists of two parts. In the first part we show that the optimality relationship between J_1 and J_2 is the same as that between I_1 and I_2. In the second part, we show that for any pair of complete plans $\mathcal{P}_1$ and $\mathcal{P}_2$, the optimality relationship between $\mathcal{P}_1$ and $\mathcal{P}_2$ is the same as that between I_1 and I_2, if $\mathcal{P}_1$ and $\mathcal{P}_2$ differ from each other only by the partial plans until O_j in the way that the partial plan in $\mathcal{P}_1$ is J_1 and in $\mathcal{P}_2$ is J_2.

Part 1: We first show that J_1 is better than J_2 if I_1 is better than I_2. According to Definition 1, there are three possible situations where I_1 can be better than I_2. For each situation, the proof to show that J_1 is better than J_2 is similar to the proof for a specific case discussed in Section 6. Hence, here we provide only references to the corresponding cases in the proof in Section 6.

- **Situation 1:** $C_b(I_1) \leq 1 < C_b(I_2)$, i.e., I_1 is feasible whereas I_2 is infeasible. The proof is similar to that for Case 1.1.
- **Situation 2:** $C_b(I_1) \leq 1$, $C_b(I_2) \leq 1$, and $C_u(I_1)$ ¡ $C_u(I_2)$, i.e., both I_1 and I_2 are feasible. The proof is similar to that for Case 1.2.
- **Situation 3:** $1 < C_b(I_1) \leq C_b(I_2)$, i.e., both I_1 and I_2 are infeasible. The proof is similar to that for Case 2.

The symmetric case that J_2 is better than J_1 if I_2 is better than I_1 can be proved in the same way. Moreover, we can easily extend the proof to show that for an operator O_j with multiple direct upstream operators, the optimal *SPE-op* rooted partial plan until O_j can be constructed from the respective optimal partial plans until each direct upstream operator of O_j.

Part 2: In this part, we show that for a pair of complete plans which are constructed as extensions of J_1 and J_2 respectively, if they differ from each other only by the partial plan J_1 and J_2, then the optimality relationship between them is the same as that between J_1 and J_2. Strictly, we need to show that the optimality is retained along the plan construction procedure until the root node of the logical plan. However, if we can prove for the direct downstream operator of O_j, which is O_k in Figure 3a, that no matter in which engine O_k is placed, the optimality relationship between the partial plans extended from J_1 and J_2 is the same as the optimality relationship between J_1 and J_2, then we can apply the same reasoning recursively. Therefore, in the following, we only show that for the two partial plan pairs (K_1, K_2) and (K_3, K_4) in Figure 3c, the optimality within each pair is the same as that between J_1 and J_2, and is therefore the same as that between I_1 and I_2.

For the pair (K_1, K_2) where O_k is assigned to the SPE, the same proof in Part 1 can be applied. The proof for the pair (K_3, K_4) is similar. Note that in the partial plans K_3 and K_4, O_k is placed in the database engine, and the source-driven input processing cost u of O_k^{db} is $\lambda_j c_k^{db}$. If the downstream operator of O_k in K_3 and K_4 is placed in the database engine as well, then the two resulting plans, say K_3' and K_4', have a composite operator $O_{k'}^{db}$. The source-driven input processing cost u' of $O_{k'}^{db}$ is $\lambda_j c_{k'}^{db}$. Although u' is different from u, u' is the same in both K_3' and K_4' and therefore does not influence the optimality relationship between K_3' and K_4'. □

Search Space Size. With the above described pruning strategy, for a logical plan with n R2R operators, we get only one *SPE-op* rooted complete execution plan, all the other plans are rooted at a *DB-op*. For logical plans containing only unary operators, we can reduce the search space size from 2^n to $n + 1$. For logical plans containing also binary operators, the search space size depends heavily on the number of binary operators in the tree; because when constructing a *DB-op*-rooted plan at a binary operator, we must consider all possibilities of combining partial plans until the left child of the operator with partial plans until the right child of the operator. In the worst case where all n R2R operators in the plan are binary operators, the logical plan is a complete binary tree. Ignoring window operators at leaf nodes, the height of the tree is $h = \lceil log_2(n + 1)\rceil$. Given the height of a binary tree, we can define the upper bound of the search space size as function of h in a recursive way:

$$f(1) = 2;\;\; f(h) = 1 + f(h-1)^2.$$

The complexity of $f(h)$ is $\mathcal{O}(f(h)) = 2^{2^{h-1}}$. By replacing h with $\lceil log_2(n+1)\rceil$, $\mathcal{O}(f(h))$ is approximately $2^{n/2}$, which is exponential. To be able to optimize queries with a large number of binary R2R operators with reasonable time, one solution is to decompose the logical plan produced in Phase-One into multiple subplans, each with a moderate number of binary operators, optimize these subplans in their post order, and construct the final execution plan by combining optimal execution plans of the subplans.

8 Evaluation

In this section, we evaluate the proposed continuous query optimizer from three aspects: the optimization time (Section 8.2), the quality of optimization results (Section 8.3), and the influence of the plan feasibility check on the quality of optimization results (Section 8.4).

8.1 Setup

We implemented the proposed optimization solution by directly extending the SQL optimizer of the columnar IMDB in our prototype. Specifically, we added the cost estimation for *SPE-ops*, and implemented the proposed two-phase optimization approach. Our system is deployed on a HP Z620 workstation with 24-cores (1.2GHz per core) and 96 GB RAM, running SUSE 11.2.

For our experiments we used the real-world energy consumption data originating from smart plugs deployed in households [JZ14]. Each smart plug is uniquely identified by a combination of a *house id*, a *household id*, and a *plug id*. Each plug has two sensors. One sensor measures the instant power consumption with *Watt* as unit; the other sensor measures the total accumulated power consumption since the start (or reset) of the sensor with *kWh* as unit. Each measurement is represented as a relational tuple. The type of the measurement is indicated by the *property* field in the tuple. Sensors report measurements every 1 second and measurements from all smart plugs are merged into a single data stream. The original rate of this sensor data stream is approximately 2000 tuples/sec. To test with higher data rates, we devised a custom program, which can replay the original sensor data at a configurable speed, simulating a higher report frequency of smart plugs.

We used the following continuous queries ($CQ1$–$CQ6$) to test our federated optimizer:

- $CQ1$**:** For each smart plug, count the number of load measurements in the last 5 minutes whose value is higher than 90% of the maximum load in the last 5 minutes.
- $CQ2$**:** For each smart plug, count the number of load measurements in the last 5 minutes whose value is higher than the average load in the last 5 minutes.
- $CQ3$**:** For each smart plug, compare the maximum and average load within the last 5 minutes with the maximum and average load within the last 1 minute.
- $CQ4$**:** $CQ4$ is similar to $CQ3$ but only compares the average load within the two different time windows.
- $CQ5$**:** For each household, find the maximum total load reported by a single smart plug within the last 5 minutes.
- $CQ6$**:** For each smart plug, compare the average loads within the last 1, 3, and 5 minutes.

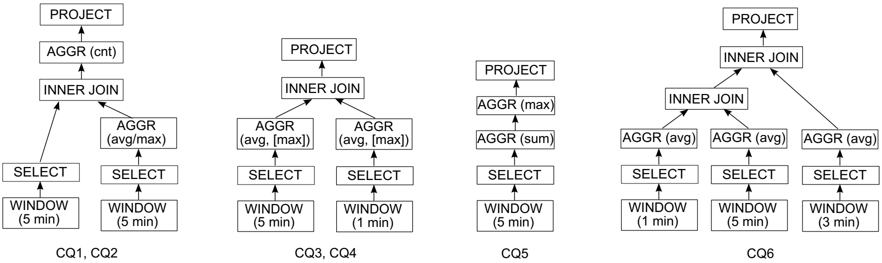


Figure 4: Logical plans of $CQ1$–$CQ6$.

All windows in these queries are time-based sliding windows and slide every 1 second. Figure 4 shows the logical query plans devised by our optimizer. We intentionally included $CQ2$ and $CQ4$ into our test, although they look similar to $CQ1$ and $CQ3$ respectively. The reason is that windowed AVG can be computed incrementally whereas windowed MAX cannot [GHM$^+$07]. Hence, the cost of AVG is normally lower than the cost of MAX in SPEs. We would like to study queries with aggregate operators of different costs.

8.2 Optimization Time

We first tested the efficiency of our optimizer in terms of the optimization time. As mentioned in Section 7, the search space size, thereby the optimization time, is heavily influenced by the number of binary R2R operators in the query. Therefore, in this experiment, we took $CQ4$ as a template and constructed multi-way join queries which compare the average loads of each smart plug within time windows of variant sizes. For instance, a 5-way join query constructed in this way first calculates the average loads of each smart plug within the last 1, 3, 5, 7, and 9 minutes, and then joins these average loads for each smart plug. In this experiment, we did not apply the query decomposition in Phase-Two as discussed in Section 7. For each query, we conducted the optimization 10 times and took the median of the measured optimization times. The results are summarized in Table 2.

Table 2: Optimization time for queries with different numbers of operators.

	2-way join	5-way join	8-way join
Opt. time of Phase-One (ms)	0.9	68.5	100.5
#R2R op. in Phase-One produced logical plan	6	15	24
#plans examined in Phase-Two w/o pruning	64	327168	16777216
#plans examined in Phase-Two with pruning	11	312	8411
Opt. time of Phase-Two with pruning (ms)	12.3	908.6	61335.5
Total opt. time (ms)	13.2	977.1	61436

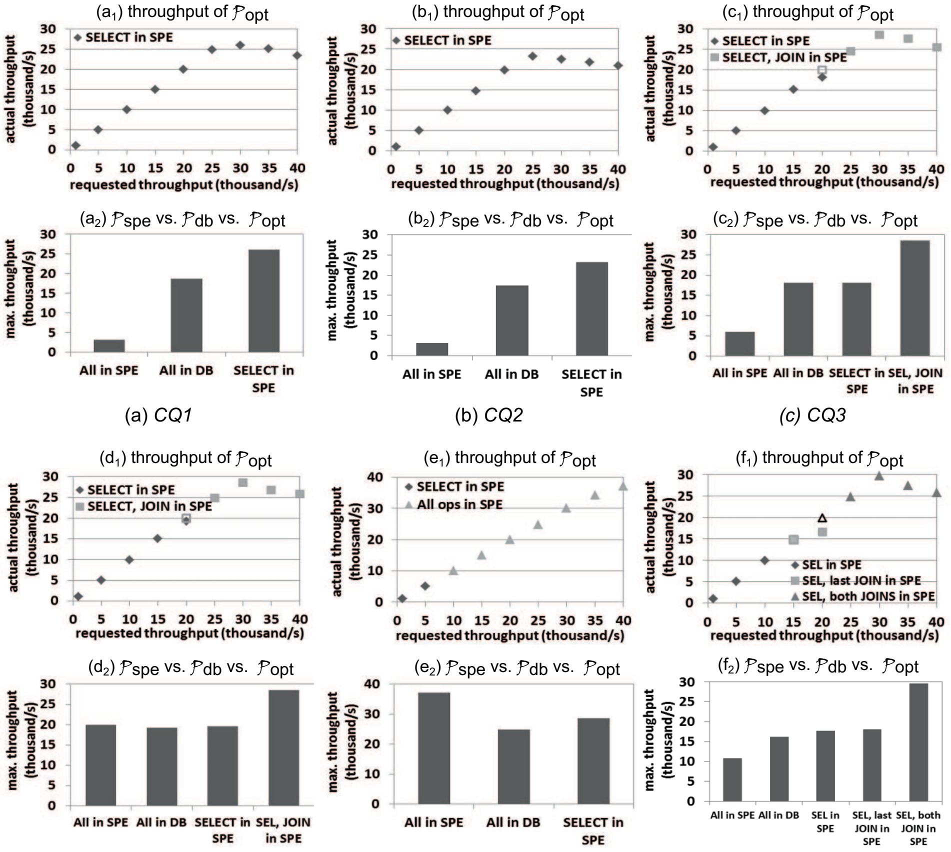
We can see from the results that with the pruning approach described in Section 7, we significantly reduced the number of plans to be examined in Phase-Two optimization. The results also suggest that in our system, it is reasonable to decompose large logical plans into subplans with 15 operators in Phase-Two. With such decomposition, the logical plan of the 8-way join query produced in Phase-One can be split into two sub-plans, thereby reducing the optimization time from 1 minute to around 2 seconds. Note that we did not provide the optimization time of Phase-Two when the pruning is deactivated, because the experiment would have taken too long and is not meaningful due to the large search space. To be complete, we list the optimization times for $CQ1$–$CQ6$ in Table 3.

Table 3: Optimization time for $CQ1$–$CQ6$.

	$CQ1$	$CQ2$	$CQ3$	$CQ4$	$CQ5$	$CQ6$
Opt. time of Phase-One (ms)	1.3	1.3	0.87	0.86	22.5	5.2
Opt. time of Phase-Two with pruning (ms)	7.7	7.5	11.4	10.8	1.3	58.9
Total opt. time (ms)	9	8.8	12.27	11.66	23.8	64.1

8.3 Effectiveness of the Federated Optimizer

Recall that our optimizer estimates costs of query plans based on data rates of source streams, and finds the optimal plan of a query based on the costs of plans (see Section 5). The data rates of source streams also define the *requested throughput* of a query. For each query in our test, we varied the rate of the sensor data stream from 1000 to 40000 tuples/sec, and asked the optimizer to produce the optimal execution plan for each data rate. For each optimal plan produced by the optimizer, we deployed it in our prototype system, pushed the sensor-data into the system at the corresponding rate, and observed the actual throughput of the plan. The results of this experiment are shown in Figure 5.



(a) *CQ1* (b) *CQ2* (c) *CQ3*

(d) *CQ4* (e) *CQ5* (f) *CQ6*

Figure 5: Performance of devised optimal plans for $CQ1$–$CQ6$ at increasing input data rates.

For $CQ1$ and $CQ2$ (see Figure 5a$_1$ and $5b_1$), for all examined data rates the optimizer picked the plan which places SELECTs in the SPE and the rest of the operators in the columnar IMDB. The reason for this optimization decision is that $CQ1$ and $CQ2$ compute correlated aggregates, which require scanning the tuples within a time window twice to compute a result. Even for a data rate of 1000 tuples/sec, a 5-minute time window contains 300k tuples. Frequent re-scanning of the window pushes the SPE to its limits. In contrast, the IMDB can compute the correlated aggregate more efficiently, despite the cost of transferring data between the two engines. The SELECTs are placed in the SPE to reduce the amount of data to be transferred to the IMDB.

To verify the superiority of our operator-level optimization approach over a query-level optimization approach, we compared the *maximum throughputs* of the optimal federated plan $\mathcal{P}_{opt}$, the pure SPE plan $\mathcal{P}_{spe}$, and the pure IMDB plan $\mathcal{P}_{db}$ for $CQ1$ and $CQ2$. We see from Figure 5a$_2$ and Figure 5b$_2$ that for both queries, the optimal federated plan can result in about 8 times higher throughput than the pure SPE plan. The maximum throughput of the pure IMDB plan is also lower than the federated plan, because it transfers more data from the SPE to the IMDB, thereby resulting in a higher cost.

For $CQ3$ (see Figure 5c$_1$), the plan which places only SELECTs in the SPE (denoted by $\mathcal{P}_{opt1}$) remains optimal until the data rate reaches 20k tuples/sec. For higher data rates, $\mathcal{P}_{opt1}$ becomes infeasible, and the plan which places both SELECTs and JOIN in the SPE (denoted by $\mathcal{P}_{opt2}$) becomes optimal. Note that when the data rate is below 20k tuples/sec, $\mathcal{P}_{opt2}$ is also feasible; however, it is not picked by the optimizer because it has higher total utilization cost than $\mathcal{P}_{opt1}$. The maximum throughputs shown in Figure 5c$_2$ confirm that $\mathcal{P}_{opt1}$ becomes infeasible at a lower data rate compared to $\mathcal{P}_{opt2}$. When the data rate is 20k tuples/sec, the actual throughput of $\mathcal{P}_{opt1}$ is indeed lower than the requested throughput, which suggests that $\mathcal{P}_{opt1}$ is already infeasible at this date rate, and $\mathcal{P}_{opt2}$ should have been chosen. The throughput of $\mathcal{P}_{opt2}$ at the rate of 20k tuples/sec is indicated by the hollow square in Figure 5c$_1$. This outcome of missing the actual optimal plan is caused by the imperfection of the cost estimation, which we believe is a common issue shared by all cost-based optimizers. However, the difference between the actual throughputs of $\mathcal{P}_{opt1}$ and $\mathcal{P}_{opt2}$ is small, and the optimizer successfully finds the correct optimal plan for all the other examined data rates. For $CQ3$, the federated plan again results in higher throughput than the pure SPE and IMDB plans.

The optimization results for $CQ4$ is similar to that for $CQ3$ (see Figure 5d$_1$). However, the pure SPE plan of $CQ4$ can support much higher data rate than the pure SPE plan of $CQ3$ (see Figure 5d$_2$), which confirms that computing MAX is more expensive than computing AVG in the SPE.

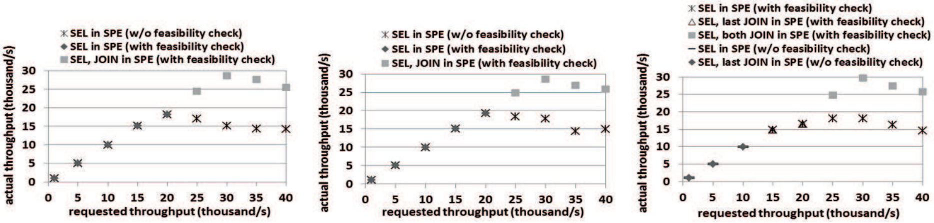
For $CQ5$ (see Figure 5e$_1$), the optimizer picked the plan which places SELECT in the SPE when the data rate is below 10k tuples/sec. For higher data rates, the total utilization cost of this plan becomes higher than that of the pure SPE plan, due to the increasing cost of data transfer between the two engines. As a result, the optimal plan switches to the pure SPE plan. Moreover, unlike for $CQ1$-$CQ4$, the pure SPE plan of $CQ5$ has higher maximum throughput compared to its federated plan alternatives (see Figure 5e$_2$).

$CQ6$ is a 3-way join query. Its optimal plan changed twice as the increase of the data rate (see Figure 5f$_1$). For data rates below 10k tuples/sec, the optimal plan has only SELECTs in the SPE (denoted by $\mathcal{P}_{opt1}$). At higher data rates until 20k tuples/sec, the optimal plan has the second JOIN operator in the SPE as well (denoted by $\mathcal{P}_{opt2}$). For even higher data rates, only the aggregation operators are left in the IMDB (denoted by $\mathcal{P}_{opt3}$). The switch from $\mathcal{P}_{opt1}$ to $\mathcal{P}_{opt2}$ was due to the higher total utilization cost of $\mathcal{P}_{opt1}$, and the switch from $\mathcal{P}_{opt2}$ to $\mathcal{P}_{opt3}$ was due to the infeasibility of $\mathcal{P}_{opt2}$ (Figure 5f$_2$). Similar to the case of $CQ3$ and $CQ3$, the optimizer missed the actual optimal plan at the rate of 25k tuples/sec, as indicated by the hollow triangle in Figure 5f$_1$.

In summary, our federated optimizer performs well with respect to the quality of optimization results. Especially, for each examined query, when the data rate of the source stream is so high that the query becomes infeasible, our optimizer is able to choose the plan which can maximize the query throughput.

8.4 Influence of the Plan Feasibility Check

Last, we studied the influence of the plan feasibility check on the quality of optimization results. To do this, we turned off the feasibility check of query plans in the optimizer, and repeated the tests described in the previous section for all six queries. For each query, we



(a) $CQ3$ (b) $CQ4$ (c) $CQ6$

Figure 6: Throughput of optimal plans devised with and without plan feasibility check.

compared the actual throughputs of the optimal plans devised with and without the plan feasibility check under each examined data rate.

The optimization results for $CQ1$, $CQ2$, and $CQ5$ without plan feasibility check are identical to that with plan feasibility check. However, for $CQ3$ and $CQ4$, without plan feasibility check, the optimizer picked the plan which places only SELECTs in the SPE at all examined data rates. However, this plan is suboptimal than the plan devised with feasibility check when the data rate is above 20k tuples/sec (see Figure 6a and 6b). For $CQ6$, without the feasibility check, the optimizer did not pick the plan which has only aggregate operators in the IMDB when the data rate is above 30k tuples/sec, which indeed has higher throughput (see Figure 6c). These results confirm the necessity of the plan feasibility check in continuous query optimization. It also implies that naive approaches for partial plan pruning without considering the plan feasibility may result in suboptimal plans.

9 Related Work

Leveraging database engines for data stream processing has been studied in a few prior works [CH10, FKC$^+$09, LGI09]. Truviso [FKC$^+$09] integrates the continuous analytics technology into a fully functional database system by executing SQL queries continuously and incrementally over data before storing the data in the database. DataCell [LGI09] is a stream processing engine built on top of MonetDB. Chen et al. [CH10] extend PostgreSQL to support stream processing. This body of work focuses on studying how a database engine can be modified to support stream processing. In contrast, our work aims to make use of the fact that SPEs and modern database systems already co-exist in many real-world setups, and proposes federated optimization and execution of continuous query to leverage the advantages of both types of systems. Our experimental results in Section 8.3 confirm the potential of federated database and stream processing systems. For the same reason, we did not follow the approach of extending the SPE directly with same implementations used in the IMDB, which can avoid the inter-system data transfer and lead to a better query processing performance. However, our query optimization approach can be applied by such extended SPEs as well to determine the best implementation alternative of each query operator from all alternatives available in the system.

MaxStream [BCD$^+$10] is a federated stream processing system which integrates multiple SPEs and databases. The federator layer is built on top of a relational database system. In MaxStream, data streams first pass through the federator layer, where the data are persisted into, or joined with static database tables if needed; subsequently, the streams are forwarded to a specific SPE for processing. However, MaxStream does not have an optimizer for continuous queries. ASPEN [LMB+10] is a project about integrating and processing distributed stream data sources in sensor devices, traditional PCs, and servers. It has a federated optimizer to assign queries across multiple subsystems. However, the optimizer does not consider the feasibility of continuous query plans, and lacks experimental support for its effectiveness. Cyclops [LHB13] integrates a centralized stream processing system (Esper [Esp]), a distributed stream processing system (Storm [Sto]), and a distributed batch system (Hadoop [Apa]) for executing continuous windowed aggregate queries. Cyclops uses black-box modeling to build cost models. Its optimizer selects the most suitable system for a given continuous query based on the window specification (range and slide) and the data arrival rate. In contrast, our optimizer works at the operator granularity, whose superiority has been confirmed by our experimental results.

Optimization of SQL queries in federated or distributed database systems [BCE+05, DH02, SL90] has been well-studied. However, existing solutions cannot be used directly for federated continuous query optimization, because they do not consider the feasibility of continuous query plans. Optimization of continuous SPJ queries concerning the plan feasibility and query throughput was initially studied in [VN02], and was extended by [AN04], which considers the optimal placement of load shedding operators in infeasible plans when computation resources are insufficient. Cammert et al. [CKSV08] deal with the similar resource management problem, and propose techniques which are based on the adjustment of window sizes and time granularities. Moreover, the cost model in [CKSV08] supports queries containing aggregation operators. However, these works do not consider query optimization in federated systems as described in this paper. There is a large body of work about operator placement in distributed or heterogeneous stream processing environments (e.g., [DLB+11] and works surveyed in [LLS08]). These works normally assume that the pipelined relationships among query operators are already determined, and consider only the placement of operators in the available processing nodes/systems. Furthermore, they do not adopt the feasibility-dependent optimization objective as we do. Nevertheless, studying how to adapt these optimization approaches in our Phase-Two optimization would be an interesting direction for future work.

10 Conclusion

In this paper, we propose a cost-based query optimization approach for federated execution of continuous queries over a SPE and a database system. To fully exploit the potential of hybrid execution of continuous queries, our optimizer works at the operator level and determines the optimal placement for each operator in a query based on the characteristics of the query and involved data streams. Moreover, the optimizer takes into account the feasibility of continuous query plans and the non-additivity of the query execution cost caused by the federation. We experimentally demonstrated the effectiveness of our optimizer in a prototype system composed of a state-of-the-art SPE and a columnar IMDB. Even for simple queries, our optimizer can make non-obvious decisions which result in up to 4 and 1.6 times higher throughput compared to the pure SPE-based execution and the pure IMDB-based execution, respectively. This result confirms that distributed execution of continuous query across SPEs and database engines is viable and promising, worthy of further exploration.

For future work, we plan to relax our assumptions on the static environment and consider runtime re-optimization in response to changing stream characteristics. Extending the current solution to support multi-query optimization (MQO) is another important direction. Apart from finding common fragments among multiple queries as is done by conventional MQO methods, we envision the sharing of inter-system data transfer channels as a particularly important aspect in the extended solution to save the communicate cost. In addition, in principle, we can add more SPEs or database engines into our system. With more than two engines, the two-phase optimization approach can still be applied; however, the pruning strategy currently applied in Phase-Two needs to be revisited.

References

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[AN04] Ahmed M. Ayad and Jeffrey F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams. In *SIGMOD*, pages 419–430, 2004.

[Apa] Apache Hadoop. `http://hadoop.apache.org/`.

[BCD+10] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Ankush Gupta, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun, and Jin Zhang. A demonstration of the MaxStream federated stream processing system. In *ICDE*, pages 1093–1096, 2010.

[BCE+05] J.A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.-C. Wu. Distributed/heterogeneous query processing in Microsoft SQL server. In *ICDE*, pages 1001–1012, 2005.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *VLDB*, 3(1-2):232–243, September 2010.

[CH10] Qiming Chen and Meichun Hsu. Experience in extending query engine for continuous analytics. In *DaWaK*, pages 190–202, 2010.

[CKSV08] Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE TKDE*, 20(2):230–245, February 2008.

[DH02] Amol V. Deshpande and Joseph M. Hellerstein. Decoupled Query Optimization for Federated Database Systems. In *ICDE*, pages 716–, 2002.

[DLB+11] Michael Daum, Frank Lauterwald, Philipp Baumgärtel, Niko Pollner, and Klaus Meyer-Wegener. Efficient and Cost-aware Operator Placement in Heterogeneous Stream-processing Environments. In *DEBS*, pages 393–394, 2011.

[Esp] Esper. `http://esper.codehaus.org`.

[FKC+09] Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *CIDR*, 2009.

[GHM+07] T.M. Ghanem, M.A. Hammad, M.F. Mokbel, W.G. Aref, and A.K. Elmagarmid. Incremental Evaluation of Sliding-Window Queries over Data Streams. *IEEE TKDE*, 19(1):57–72, 2007.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[HLY$^+$09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[HM94] Waqar Hasan and Rajeev Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *VLDB*, pages 36–47, 1994.

[HS91] Wei Hong and Michael Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, pages 218–225, 1991.

[Ji13] Yuanzhen Ji. Database support for processing complex aggregate queries over data streams. EDBT Workshop, pages 31–37, 2013.

[JZ14] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 Grand Challenge. In *DEBS*, pages 266–269, 2014.

[KS04] Jürgen Krämer and Bernhard Seeger. PIPES: a public infrastructure for processing and exploring streams. In *SIGMOD*, pages 925–926, 2004.

[KS09] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM TODS*, 34(1):4:1–4:49, April 2009.

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. Exploiting the power of relational databases for efficient stream processing. In *EDBT*, pages 323–334, 2009.

[LHB13] Harold Lim, Yuzhang Han, and Shivnath Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.

[LLS08] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement Strategies for Internet-Scale Data Stream Systems. *IEEE Internet Computing*, 12(6):50–60, November 2008.

[LMB$^+$10] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. SmartCIS: integrating digital and physical environments. *SIGMOD Record*, 39(1):48–53, 2010.

[Pro] SAP Event Stream Processor. `http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html`.

[PS06] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *EDBT*, pages 445–464, 2006.

[SAC$^+$79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[SFGL13] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. SAP HANA: The Evolution from a Modern Main-memory Data Platform to an Enterprise Application Platform. *Proc. VLDB Endow.*, 6(11):1184–1185, August 2013.

[SL90] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990.

[Sto] Storm. `http://storm-project.net/`.

[VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.