# Incremental Stream Query Merging

Ankit Chaudhary[1], Jeyhun Karimov[2], Steffen Zeuch[1,2], Volker Markl[1,2]
Technische Universität Berlin[1],DFKI GmbH[2]
firstname.lastname@(tu-berlin.de[1], dfki.de[2])

## ABSTRACT

Stream Processing Engines (SPEs) execute long-running queries on unbounded data streams. They mainly focus on achieving high throughput and low-latency for a single query. This focus neglects the possible sharing opportunities of data and compute among multiple, long-running queries. Common approaches in batch-oriented systems mainly utilize simple and fast query merging algorithms based on syntactic similarities as the overhead of more extensive approaches would not amortize over the short query runtime. In contrast, streaming queries are continuous and long-running, such that extensive approaches, like taking the *semantics* of queries into account, may pay off. Furthermore, the long-running nature of streaming queries requires the merging of existing and newly arriving queries, unlike batch queries where merging is performed only among a batch of arriving queries.

In this paper, we propose Incremental Stream Query Merging (ISQM), an end-to-end solution to identify and maintain sharing among thousands of stream queries. ISQM captures the semantic information of stream queries to enable merging even in the presence of syntactic differences. Our evaluation shows that ISQM exploits up to 65x more sharing opportunities than the naive baseline using hash-based signatures, scales linearly for thousands of queries, and saves a significant amount of resources compared to state-of-the-art approaches.

## 1 INTRODUCTION

A wide variety of stream processing engines (SPEs), e.g., Flink, Spark, or Storm, have been proposed to address the needs of modern data processing applications in the area of mobility, health care, or manufacturing [29, 44]. Cloud vendors offer managed stream analytics services that address the needs of millions of users [33, 35, 42]. The resulting workloads contain thousands of potentially overlapping queries and thus have a high potential to reduce redundant computation. A recent study revealed that over 45% of the daily data processing jobs submitted by approximately 65% of the users have commonalities, resulting in millions of sub-expression overlaps [23].

Despite this trend, commercial SPEs focus mainly on achieving high throughput and low-latency processing for individual queries [20, 25]. Some guidelines even suggest exclusively running one compute cluster per stream query [12]. This focus neglects the possible resource efficiency improvements from sharing data and compute resources among long-running queries. In the future, redundancy reduction will be the key enabler to achieve high scalability for real-time applications [4, 27]. Potential applications where such optimizations provide benefits are traffic congestion detection, fleet management, or surveillance within a smart city [13, 30, 41].

As motivation, consider real-time analytics over traffic streams within a smart city. In particular, traffic analysts utilize specialized

```
Q1: QUERY::FROM("TRAFFIC").MAP(SPEED=SPEED*1.6)
        .FILTER(SPEED>100 && COLOR == 'RED')
        .TUMBLINGWINDOW(MINUTES,5).KEYBY("COLOR")
        .APPLY(MAX(SPEED));
Q2: QUERY::FROM("TRAFFIC").FILTER(SPEED*1.6>100)
        .MAP(SPEED=SPEED*1.6)
        .SLIDINGWINDOW(MINUTES,5,5).KEYBY("COLOR")
        .APPLY(MAX(SPEED)).FILTER(COLOR == 'RED');
```

**Listing 1: Two equivalent queries to find in every 5 min a red vehicle having max speed more than the speed limit.**

frameworks (e.g., FLOW [38]) to monitor suspicious activities. Such queries help prevent or detect untoward situations during a public gathering or near sensitive installations [31, 34]. To this end, it is imperative to allow efficient resource utilization that enables large-scale deployment of such queries. Such specialized frameworks support declarative query languages that allow analysts to define syntactically different queries that produce the same result, i.e., semantically equivalent. This declarativity within query languages introduces additional complexity while identifying sharing opportunities among deployed queries to reduce redundant data processing and transmission.

Listing 1 shows two syntactically distinct but semantically similar stream queries for traffic surveillance of suspicious vehicles. Q1 first transforms speed into miles per hour (mph) and then selects red-colored vehicles with a speed above 100 mph. On the resulting stream, Q1 then applies a 5 min tumbling window to group vehicles by their color and find the maximum speed keyed per group. In contrast, Q2 first selects vehicles with a speed above 100 mph and then transforms their speed. On the resulting stream, Q2 applies a sliding window of 5 min length and slide (equivalent to a 5 min tumbling window) to find a vehicle with maximum speed keyed by their color and selects only red colored vehicle. Despite being syntactically different (i.e., applying different operators in a different order), both queries produce the same result stream.

On the one hand, research proposes query containment [21, 43, 45, 49] as a means to merge stream queries. Containment-based approaches do not perform well for queries with structural and syntactic differences as they require queries to be structurally similar (i.e., apply the same operator order) [21, 49]. On the other hand, current approaches that allow sharing identification among structurally different queries, such as SPES [48] or EQUITAS [47], are designed for batch queries. Stream queries introduce unique challenges, such as special semantics, ad-hoc submissions, and long-running lifetime.

The unique challenges for stream queries in an SPE that aims to exploit sharing potential among thousands of queries are as follows. First, an SPE should be able to handle the special semantics of stream operators and exploit their unique sharing opportunities. In particular, stream queries might contain special window semantics to define joins and aggregations over an unbounded stream of data. Second, an SPE should consider both running and newly arriving queries for sharing identification. Unlike batch queries, stream queries are usually long running and thus, the system needs an efficient mechanism to identify sharing opportunities among them. Third, an SPE should be able to efficiently represent and maintain sharing opportunities for thousands of stream queries during run-time by using highly efficient data structures. Combining these factors with a large volume of structurally distinct stream queries invites a fresh look into sharing identification and representation in this unique setup.

In this paper, we propose Incremental Stream Query Merging (`ISQM`), an end-to-end approach to identify and maintain sharing opportunities among thousands of concurrent stream queries. `ISQM` utilizes special signatures that allow it to capture operations performed by both relational and special stream operators, such as window joins or aggregations. As these signatures capture semantic information, `ISQM` can identify sharing opportunities even in the presence of syntactic and structural differences. In addition, this allows `ISQM` to act as a preprocessing step for containment-based approaches in the presence of structurally distinct queries. `ISQM` maintains a Global Query Plan (GQP) that enables exploration of sharing opportunities among both newly arriving and already running queries. In addition, `ISQM` enables low-latency sharing identification by using special indexes to prune the search space for sharing identification. Overall, `ISQM` uses heuristics, candidate pruning, and adaptive selection of sharing identification strategies to trade off optimization time and sharing identification efforts. `ISQM` identifies sharing opportunities among thousands of stream queries, independent of their syntactical structure, and efficiently manages thousands of shared query plans. Our experiments show that `ISQM` finds up to 65x more sharing opportunities, within a reasonable amount of time. Furthermore, the exploitation of sharing opportunities in `ISQM` leads to significant resource savings during run-time (up to 5.4x less compute and 16x less network resources).

In summary, we propose `ISQM`, a novel end-to-end solution for scalable query merging in SPEs, with the following contributions:

(1) We propose a new approach to capture transformations within a streaming data flow using signatures.
(2) In addition to complete sharing, `ISQM` enables the exploitation of partial sharing opportunities across stream queries.
(3) We propose SM+ which is a novel adaptive technique that seamlessly switch between a fast and a more extensive sharing identification based on the query workload.
(4) We introduce a solution to efficiently maintain identified sharing under continuous arrival and removal of queries.

We organize remainder of the paper as follows: Sec. 2 presents the relevant background. Sec. 3 describes the system overview of `ISQM`, Sec. 4 presents the signature computation and representation, and Sec. 5 introduces the internal representation of our global query plan. Then, in Sec. 6, we outline our approaches to identify sharing and present details on managing evolving global query plan in Sec. 7. Finally, we evaluate our approach in Sec. 8, present related work in Sec. 9 and conclude in Sec. 10.

## 2 BACKGROUND

**Stream Processing.** Long-running stream processing queries allow for real-time analytics over an unbounded stream of tuples. To discretize an unbounded stream, stream queries employ special stream semantics, i.e., *Windows*, to process a collection of tuples together. A window $W$ defines potentially overlapping subsequences of stream tuples and is commonly characterized by window *type* (e.g., tumbling, sliding, or session) and by *measure* (time-based or count-based) [40]. A window type defines the frequency and the length of a window. Tumbling and sliding windows define the frequency using a slide size ($l_s$) and a length

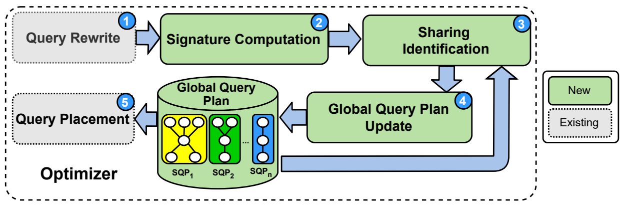


**Figure 1: System overview of `ISQM`.**

($l$). A tumbling window is a particular case of a sliding window where the slide size and length are the same, i.e., $l_s = l$. In contrast, a sliding window with $l_s < l$ creates overlapping subsequences from a stream. A window measure determines the termination of an active window. A time based window uses timestamps ($m_{ts}$) to determine the termination of a window, i.e., $m_{ts} > l$.

**Satisfiability Modulo Theory.** Satisfiability Modulo Theory (SMT) is used to determine if a mathematical formula is satisfiable. The software verification community uses SMT solvers [10] to automate the verification and validation of complex programs [1, 11]. They convert these complex programs into *Symbolic Representations* (SRs), which represent first-order logic formulas. In particular, an SR is an expression consisting of various arithmetic, logical, constant, or variable symbols. To solve the formula, an SMT solver substitutes different values in the variables of the SR and determines its satisfiability. For example, the SR "$2y=y+y$" contains a logical expression "$=$", an arithmetic expression "$+$", a constant value "2", and a variable "$y$". A SMT solver substitutes "$y$" with different constant values to determine the expression's satisfiability (e.g., when $y$=1). In particular, the SMT solver returns *satisfiable* if it finds a solution, otherwise, it returns *unsatisfiable*. For example, it determines that the SR ($x > 0 \wedge x < 0$) is unsatisfiable as there exists no value of $x$ for which the SR satisfies. Additionally, we can configure an SMT solver with a predefined time budget after which it terminates and marks the expression as unsatisfiable.

## 3 SYSTEM OVERVIEW

This section gives an overview of `ISQM`. Figure 1 shows how `ISQM` extends the query optimizer of an SPE with additional phases to identify and exploit sharing opportunities across queries. In essence, `ISQM` operates after optimizing query plans (e.g., applying rewrite rules) and before mapping queries to nodes in a cluster (e.g., operator placement [5]). For query merging, `ISQM` introduces three phases, i.e., Signature Computation, Sharing Identification, and Global Query Plan Update. In Figure 1 we highlight these new phases in green and the existing phases in grey. In a preliminary step (not shown), an SPE transforms a submitted query into a graph of connected operators, the so-called query plan, and delivers this plan to the optimizer. Overall, `ISQM` is a general-purpose framework that allows easy integration of other sharing identification approaches, e.g., structural analysis of query plans ([14, 19, 24]) or signatures-based analysis on syntactic properties of queries ([6, 23, 27]).

In the following, we will describe the optimization process in detail. First, the *Query Rewrite* phase ① applies a set of rules to normalize input query plans. This normalization eliminates some syntactic differences (e.g., redundant expression elimination) and optimizes query plans (e.g., join-order optimization). The rewrite rules vary across different systems and query plans [32]. Second, the query plan enters our *Signature Computation* phase ②. In this phase, `ISQM` constructs a set of signatures (`SIGs`). In particular, this phase traverses the query plan bottom-up, i.e., from source to the sink operator. For each operator, the phase creates a signature based on the operator properties and the signatures of its upstream operators. A common approach is to compute signatures based on the hash of syntactic properties of operators [26, 27]. We call these approaches sharing identification based on hash-based signatures (HB). However, such strategies identify limited sharing opportunities and rely on rewrite rules to resolve syntactic ambiguities among the queries. In ISQM, the signature computation phase captures all syntactic and semantic transformations within a streaming data flow and represents signatures using SRs. As a result, a signature covers all semantic transformations performed by the operator and its upstream operators. We present more details on these signatures and their representation in Sec. 4.

Third, the query plan enters our Sharing Identification phase 3. In this phase, ISQM uses the signatures of newly submitted and already active queries to identify sharing opportunities. In contrast, state-of-the-art approaches for batch systems consider only new queries for sharing identification ([14, 15, 47, 48]). ISQM constantly maintains *Global Query Plan* (GQP) that represents all running and merged queries. When a query plan enters the GQP, we refer to it as a *Shared Query Plan (SQP)*. We present internal representations of GQPs in Sec. 5. Our sharing identification phase consists of two steps. First, it extracts a collection of *candidate* SQPs from the GQP and considers them for exploring sharing opportunities. This extraction allows ISQM to reduce the overall search space for sharing identification at the cost of reduced sharing opportunities. Second, this phase checks for equality across candidate SQPs and the new query plan using the signatures the captures not just syntactic but also semantic information, a collection of heuristics, and an SMT solver. We refer this approach of sharing identification as Semantic-based-query-Merging approach (SM). Note that the sharing identification phase can be extended to implement other strategies that use structural analysis instead of signatures (e.g., graph-isomorphism techniques) on query plans for sharing identification. We call these approaches sharing identification based on structural analysis (SA) in the remainder. If the phase finds a sharing opportunity, it forwards a pair of matched operators to the next phase. Otherwise, ISQM forwards the entire new query plan to the next phase. We present more details about how ISQM's sharing identification phase works in Sec. 6.

Fourth, the query plans enter our *Global Query Plan Update* phase 4. Based on the result of the previous phase, this phase updates the existing SQP (in case of a match) or adds a new SQP based on the query plan. This phase is also responsible for updating an SQP when a query leaves the system. The phase marks the resulting SQP for placement and deployment. We present details on the global query plan update phase in Sec. 7. Finally, the *Query placement* phase 5 receives the updated SQP and performs the operator placement for its execution [17].

Overall, the goal of ISQM is to enable computation and data sharing among new query plans and already running SQPs. As a result, existing resources are better utilized, and the overall efficiency of an SPE improves. In the following sections, we describe the newly introduced phases in detail.

## 4 SIGNATURE COMPUTATION PHASE

In this section, we present the *Signature Computation Phase* (SCP) of ISQM. This phase captures semantic information of operators in a query plan and represents the information as signatures. It supports full ANSI SQL syntax for stream queries. Supporting UDF [18] signatures is out of the scope of this paper. We describe the internal representation of signatures in Sec. 4.1 and show how SCP constructs signatures for different operators in Sec. 4.2.

### 4.1 Signature Representation

ISQM uses signatures to represent the semantics of operators in stream queries. These operators apply predicates, perform transformations, and compute windowed operations on tuples from upstream operators. Similar to [16, 36, 47], ISQM extracts interesting properties from these operators and computes a signature based on them. To this end, ISQM represents a signature using the following triplet: $\text{SIG}=(\text{PRED},\vec{\text{TT}},\vec{\text{WIND}})$.

PRED represents the SR for all predicates of an operator and its upstream operators. In particular, it captures the conjunction of all predicates that an output tuple has to satisfy.

$\vec{\text{TT}}$ represents a vector of tuple transformations ($tt$)s. A $tt$ captures all manipulations on a stream tuple using SRs. For example, a Map updates an attribute, a Project removes or renames a set of attributes, and a Join merges attributes from two potentially distinct streams. Thus, a $tt$ maps attribute names *(where a transformation is applied)* to corresponding SR *(what transformation is applied)*. Note that, binary operators receive tuples from multiple streams, thus we store $tt$ in a vector $\vec{\text{TT}}$, i.e., one entry per stream.

$\vec{\text{WIND}}$ represents a vector of SRs, one for each window operator observed until and including the current operator. Each window operator contains a window definition and either a transformation (Window Aggregation) or a predicate (Window Join). An SR for a window operator represents conjunction of window definition (i.e., window size, slide, optional key-by attribute, time attribute) and its operation. In this paper, we focus on sliding and tumbling windows with time as the measurement.

In contrast to existing approaches designed for batch queries [47, 48], our approach captures special window semantics in query signatures. Figure 2 presents an example query and shows corresponding signatures in a purple box. Throughout this section, we will use this example query to introduce SCP.

### 4.2 Signature Computation

In general, SCP computes and assigns signatures to each operator in a query plan. This signature assignment to individual operators in a query plan allows for identifying sharing opportunities even among partially equal queries. To this end, SCP traverses the query plan operators in an upstream-to-downstream fashion and performs the following three steps for signature computation. First, SCP extracts information about the operations performed by the current operator, such as transformations, predicate evaluations, or window operations, and computes SRs based on them. Figure 2 shows these SRs in the purple rectangles connected to an operator. This step captures all the operations performed by an operator. Second, SCP updates these SRs by substituting transformations from the signatures of the upstream operators. Figure 2 shows the relationship between upstream signatures and an operator's signature using the direct dotted arrow. This step allows the updated SRs to represent the operations performed on the transformed attributes. Third, SCP constructs a new signature by combining the updated SRs with the signatures from its upstream operators. This step allows a signature to collectively represent semantic transformations performed by an operator and its upstream operators. As a result, SCP reuses newly computed signatures when computing the signatures of downstream operators.

Extracting information from an operator into SRs (first step) and merging updated SRs with upstream signatures (third step) is operator-dependent. However, the step to substitute SRs with transformations from upstream operators (second step) is common across operators. To this end, we describe the second step in the following and refer to it as the SRSubstitution process in the remainder of the paper. The SRSubstitution process takes as input an SR and the $\vec{\mathtt{TT}}$s from its upstream signatures. For each *tt* in a $\vec{\mathtt{TT}}$, SRSubstitution updates an SR by substituting its symbolic attribute with the transformation from the *tt*. The updated SR constitutes the combination of the operator's and its upstream operator's semantic information. Finally, SRSubstitution returns a collection of updated SRs, one for each *tt*.

Next, we present in detail how SCP computes SRs for an operator, updates SRs by calling SRSubstitution, and combines them with signatures from upstream operators. We first discuss unary operators and then extend the discussion for binary operators.

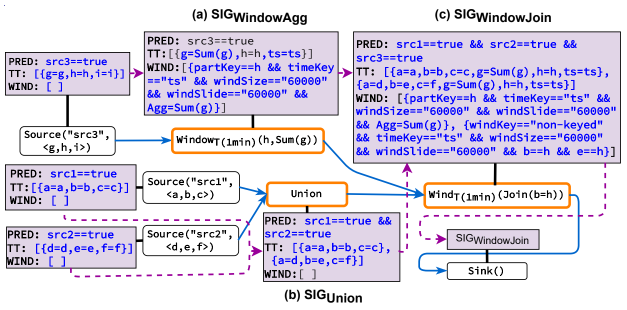
*4.2.1* ***Operators Shared With Batch Processing Systems.*** In the following, we briefly discuss signature computation for the source, map, filter, project, and sink operators.

A Source operator contains the stream name and the schema of the tuples it produces. In the first step, SCP extracts the unique stream name and schema from the source operator. As a source operator contains no upstream operators, SCP skips the remaining two steps and uses the extracted information to construct the signature. In particular, SCP computes a PRED representing an equality predicate on the stream name and a $\vec{\mathtt{TT}}$ based on source schema. SCP then initializes the signature with the PRED and $\vec{\mathtt{TT}}$ and leaves the $\vec{\mathtt{WIND}}$ empty.

A Map operator adds or updates an existing tuple attribute by applying a transformation expression. A transformation expression represents the semantic changes that the operator performs on an attribute of a tuple. To compute the signature, SCP extracts the assignment expression and computes an SR for it. Then, SCP calls SRSubstitution to update the newly computed SR using the *tts* from the upstream signature's $\vec{\mathtt{TT}}$. After that, SCP constructs a new signature by combining the updated SRs with the signature of its upstream operator. To this end, SCP assigns each updated SR to the assignee attribute in the *tts* from the upstream signature. Finally, SCP assigns the updated $\vec{\mathtt{TT}}$ and the upstream signature's PRED and $\vec{\mathtt{WIND}}$ to the new signature.

A Filter operator applies predicates on tuples from an upstream operator and outputs the tuples that satisfy the predicate. To compute the signature, first, SCP constructs an SR for the filter predicate. Second, SCP calls SRSubstitution to update the newly constructed SR using *tt*'s from the upstream signature's $\vec{\mathtt{TT}}$. SCP unifies all updated SRs by computing the disjunction among them. This unified SR represents the filter predicate applied on tuples from different upstreams. Third, SCP computes a new signature by assigning the conjunction of the unified SR from previous step and the upstream signature's PRED to the current PRED. Additionally, SCP assigns the $\vec{\mathtt{TT}}$ and the $\vec{\mathtt{WIND}}$ of the upstream signature to the new signature.

A Project operator can prune or rename the tuple attributes from a stream. The operator defines a list of attributes to retain and optionally their new attribute names. First, SCP extracts the list of attributes and their new names from the project operator. SCP skips the second step to preserve the transformations from upstream operators. Third, SCP computes a new $\vec{\mathtt{TT}}$ by pruning upstream $\vec{\mathtt{TT}}$ for attributes not in the projected attribute list and renaming the retained attribute names to corresponding new names (without changing SRs, i.e, only keys are changed). SCP



(a) SIG_WindowAgg
(c) SIG_WindowJoin
PRED: src3==true
TT: [{g=g,h=h,i=i}]
WIND: [ ]
PRED: src3==true
TT:[{g=Sum(g),h=h,ts=ts}]
WIND:[{partKey==h && timeKey =="ts" && windSize=="60000" && windSlide=="60000" && Agg=Sum(g)}]
PRED: src1==true && src2==true && src3==true
TT: [{a=a,b=b,c=c,g=Sum(g),h=h,ts=ts}, {a=d,b=e,c=f,g=Sum(g),h=h,ts=ts}]
WIND: [{partKey==h && timeKey=="ts" && windSize=="60000" && windSlide=="60000" && Agg=Sum(g)}, {windKey=="non-keyed" && timeKey=="ts" && windSize=="60000" && windSlide=="60000" && b==h && e==h}]
Source("src3", <g,h,i>)
Window_T(1min)(h,Sum(g))
PRED: src1==true
TT:[{a=a,b=b,c=c}]
WIND: [ ]
Source("src1", <a,b,c>)
Union
Wind_T(1min)(Join(b=h))
PRED: src2==true
TT: [{d=d,e=e,f=f}]
WIND: [ ]
Source("src2", <d,e,f>)
PRED: src1==true && src2==true
TT: [{a=a,b=b,c=c}, {a=d,b=e,c=f}]
WIND:[ ]
SIG_WindowJoin
Sink()
(b) SIG_Union

**Figure 2: Signature creation example.**

then computes a new signature by using the new $\vec{\mathtt{TT}}$ and retaining the PRED amd $\vec{\mathtt{WIND}}$ from upstream signature.

A Sink operator defines the configurations and location where output tuples are written. In our approach, we skip merging sink operators to allow writing output tuples for all submitted queries. To this end, SCP ignores sink operators for signature computation and instead assigns the signature of its upstream operator to it.

*4.2.2* ***Window Aggregation.*** A stream query defines a Window Aggregation operator to perform aggregations over unbounded stream of data. To this end, a window aggregation operator contains a window definition (to prepare the collection of tuples) and an aggregation function (to apply a processing on the collection). A window definition contains the type, the measure, and the *partition key* (indicating if results are grouped). An aggregation function defines an aggregation type (e.g., MIN, MAX, or AVG) and the attribute to aggregate.

To compute the signature of a Window Aggregation operator, SCP computes SRs for the window definition and aggregation function. In particular, SR combines its type, measure, partition key, and converts window measures into milliseconds to represent time under a common unit. If a partition key is not present, SCP uses "non-keyed" as the partition key in the SR. Additionally, SCP computes a separate SR for the aggregation function. Second, SCP calls SRSubstitution to update all SRs using transformations from its upstream operator. Third, SCP constructs the signature by combining the updated SRs with the upstream signature. In particular, SCP assigns the updated SRs of the aggregation function to the respective aggregation attribute in *tts*. As Window Aggregation operator only outputs the partition key, the time key (for time measure), and the aggregation attribute, SCP removes from each tt all other attribute entries. Thus, SCP computes a new $\vec{\mathtt{TT}}$ containing these updated tts. Next, SCP computes the conjunction of all updated SRs and assigns the new SR to $\vec{\mathtt{WIND}}$ from upstream signature. Finally, SCP assigns the updated $\vec{\mathtt{WIND}}$, the updated $\vec{\mathtt{TT}}$, and the upstream signature's PRED to the new signature.

Stream queries might contain multiple cascading windows. Our approach encapsulates this case by extending the window signature to include tuple information. This approach allows us to calculate the outcome of any aggregate window, without knowing the actual values of tuples. For simplicity, we skip the cascading window part in our explanation and focus on the signature of a single window.

*Example.* Figure 2(a) shows the $\mathtt{SIG}_{\mathtt{WindowAgg}}$ for the Window Aggregation operator. SCP computes the updated SRs Sum(g) for the aggregation function and (partKey==h && timeKey ==ts && windSize==60000 && windSlide== 60000) for the window definition. SCP then updates the $\vec{\mathtt{WIND}}$ of the upstream signature by adding to it the conjunction of both updated SRs. Similarly, SCP updates the $\vec{\mathtt{TT}}$ of the upstream signature with the SR for the aggregation function, the partition key h, and the time key ts. Finally, SCP assigns the updated $\overrightarrow{TT}$, updated $\overrightarrow{WIND}$, and the upstream signature's PRED to the new signature.

*4.2.3* ***Union.*** A stream query defines a Union operator to interleave tuples from two input streams and produce a single unified stream. Tuples from both input streams have to have the same schema and thus, their attribute number and types match. However, attribute names can potentially differ among schemas. Similar to existing work [39], SCP chooses the attribute names from its left upstream operator. To this end, the union operator renames the attributes from the right upstream operator to match with the attribute names from the left upstream operator.

To construct the signature of a Union, SCP skips the first and second steps as the operator performs no filter or transformation. In the third step, SCP combines the signatures from both upstream operators to compute a new signature. In particular, SCP initializes a new $\overrightarrow{TT}$ using the $\overrightarrow{TT}$ of the left upstream signature. This initialization captures that the Union operator performs no transformations on the left upstream tuples. Next, SCP updates the $\overrightarrow{TT}$ of the right upstream signature by replacing its keys with the attribute names from the left upstream operator's schema. Thus, SCP merges the newly constructed and the updated $\overrightarrow{TT}$s. Merging these $\overrightarrow{TT}$*s* captures that a Union operator receives tuples from two different streams. After that, SCP combines the PREDs and $\overrightarrow{WIND}$s from both the upstream signatures. In particular, SCP computes the conjunction of the two PREDs and merges the two $\overrightarrow{WIND}$s from both upstream signatures. Finally, SCP computes a new signature by assigning to it the merged $\overrightarrow{TT}$, the conjunction of PREDs, and the merged $\overrightarrow{WIND}$s.

*Example.* Figure 2(b) shows the $\mathtt{SIG}_{\mathtt{Union}}$ of the Union operator. SCP computes a new $\overrightarrow{TT}$ by merging $\overrightarrow{TT}$s from both upstream signatures and replacing the keys in the $\overrightarrow{TT}$ from the right upstream signature with the attribute names from the left tuple schema, i.e., [{a=a,b=b,c=c},{a=d,b=e,c=f}]. Next, SCP computes a conjunction of PREDs and merges $\overrightarrow{WIND}$s from both upstream signatures. Then, SCP assigns the new $\overrightarrow{TT}$, the conjunction of PREDs, and the merged $\overrightarrow{WIND}$s to the new signature.

*4.2.4* ***Window Join.*** A stream query defines a Window Join operator to perform joins over unbounded stream of data. To this end, a window join operator first defines windows over streams from two upstream operators and then a join predicate to join tuples from both streams from the same window.

To compute the signature, SCP extracts the window definition and the join predicate from the operator. Then, SCP constructs SRs for the window definition and the join predicate. Second, SCP calls SRSubstitution to update all computed SRs with transformations from both the upstream operators. Third, SCP constructs a new signature by combining the updated SRs with the upstream signatures. In particular, SCP computes a new $\overrightarrow{TT}$ by performing a cartesian product between $\overrightarrow{TT}$s from both upstream signatures. The cartesian product allows the construction of *tt*s, which captures transformations in all possible combinations of output tuples. Next, SCP computes a new $\overrightarrow{WIND}$ by merging the $\overrightarrow{WIND}$s from both upstream signatures. Additionally, SCP computes the conjunction of all updated SRs from the step two and adds it to the $\overrightarrow{WIND}$ of the upstream signature. Finally, SCP assigns to the new signature the newly computed $\overrightarrow{WIND}$, the newly computed $\overrightarrow{TT}$, and the conjunction of PREDs from both upstream signatures.

*Example.* Figure 2(c) shows the $\mathtt{SIG}_{\mathtt{WindowJoin}}$ for the Window Join operator. Here, $\overrightarrow{TT}$ represents cartesian product between

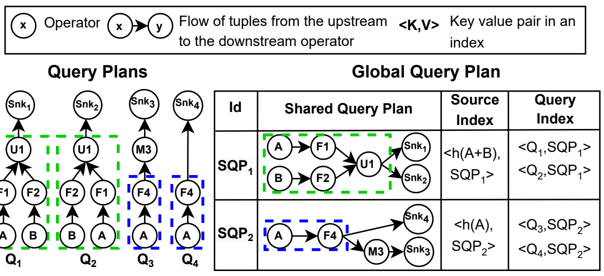


**Figure 3: An example Global Query Plan.**

both $\overrightarrow{TT}$s from the upstream signatures, i.e., [{a=a, b=b,c=c,g=Sum(g),h=h,i=i},{a=d,b=e,c=f,g=Sum(g),h=h,i=i}]. PRED represents the conjunction of PREDs from both upstream signatures. $\overrightarrow{WIND}$ adds $\overrightarrow{WIND}$s from both upstream signatures and an entry representing the conjunction of SRs for window definitions and the join predicate.

## 5 GLOBAL QUERY PLAN

This section presents the internal structure of our Global Query Plan (GQP) that represents all running queries in an SPE. ISQM utilizes the GQP for two purposes: 1) representing sharing opportunities among running queries and 2) accessing running queries for future sharing identification. To this end, a GQP consists of a collection of disjoint edges (so-called *Shared Query Plans* (SQPs)), a *Source Index*, and a *Query Index*.

**Shared Query Plan.** An SQP represents a collection of query plans such that all common operators among the query plans are merged. The resulting SQP enables data sharing (queries on the same streams) and compute sharing (queries with semantic equivalence among operators). We present the sharing identification process in detail in Sec. 6. As a design decision, we avoid merging all queries into a single SQP to prevent creating one large SQP containing potentially thousands of operators. In particular, a large SQP results in a slower sharing identification and update process due to the large number of operators. To address this issue, ISQM explores sharing opportunities for new queries only among SQPs that operate on *exactly* the same streams. This also allows a fast and parallel sharing identification at the cost of reduced sharing opportunities.

*Example.* Figure 3 shows four example query plans on the left and the GQP representing these example queries on the right. Queries $Q_1$ and $Q_2$ share common source stream (i.e. A and B) and operators as shown in green rectangle in the left of the figure. ISQM computes $SQP_1$ by merging shared operators as shown in the green rectangle on the right side of the figure. Similarly, ISQM computes $SQP_2$ by merging shared operators between $Q_3$ and $Q_4$ (shown in blue). Note that, even though $Q_1$, $Q_2$, $Q_3$, and $Q_4$ share a common stream A, ISQM does not merge these queries together.

**Source Index.** A source index maps an order-independent hash of stream names to the SQPs that operate on those streams. To find sharing opportunities, ISQM locates *candidate* SQPs operating based on the streams in the new query. A source index enables the following advantages: 1) it allows pruning the search space during sharing identification, and 2) it enables combining other interesting properties as search keys. For example, in a multi-tenant environment, index keys can combine tenant ids with stream names into a composite key. This composite key prevents merging queries belonging to distinct tenants and provides isolation during execution. Overall, the source index allows ISQM to linearly scale in the presence of thousands of running queries.

*Example.* Figure 3 shows the source index of the example GQP. The source index contains entries for both $\texttt{SQP}_1$ and $\texttt{SQP}_2$. As $\texttt{SQP}_1$ operates on sources A and B, the source index contains mapping between hash value represented by h(A+B) to $\texttt{SQP}_1$. Similarly, hash value h(A) for source A is mapped to $\texttt{SQP}_2$.

**Query Index**. A query index maps the unique identifier of a query to the SQP that contains it. Because GQP potentially contains a large collection of SQPs involving thousands of queries, ISQM uses a query index to fetch the respective SQP.

*Example.* Figure 3 shows the query index of the example GQP. The SQP $\texttt{SQP}_1$ contains queries $\texttt{Q}_1$ and $\texttt{Q}_2$. Thus, query index contains entries mapping both queries with $\texttt{SQP}_1$. Similarly for queries $\texttt{Q}_3$ and $\texttt{Q}_4$, the query index contains entries mapping them with SQP $\texttt{SQP}_2$.

## 6 SHARING IDENTIFICATION PHASE

In this section, we present the sharing identification technique that we apply in ISQM. Sec. 6.1 presents how ISQM performs equality between a pair of signatures. Sec. 6.2 presents the overall process of identifying complete and partial sharing among queries. Sec. 6.3 presents two variants that improves optimization time for sharing identification.

### 6.1 Signature Equality

To identify sharing among queries, ISQM checks equality among operators in their respective query plans. Two operators are equal if they output same tuples. A naive approach for checking equality between two operators involves checking types, expressions, and the order of the operators and their upstream operators. However, operators that have different expressions and upstream operator orders can still have semantic equivalence [47, 48] (e.g., Listing 1 shows two syntactically distinct but semantically equal queries). To identify such equivalence, ISQM utilizes signatures that capture the overall semantic information of an operator and its upstream operators. The resulting signature identifies equality independent of expressions and upstream operator orders.

ISQM utilizes a set of heuristics and an SMT solver to detect equality among two signatures. Figure 4 shows the different steps involved in the signature equality check. To use an SMT solver, ISQM computes a formula representing inequalities among SRs from a pair of signatures, i.e., $\texttt{SR}_{\texttt{SIG1}} != \texttt{SR}_{\texttt{SIG2}}$. Note that ISQM does not perform an equality check among SRs, i.e., $\texttt{SR}_{\texttt{SIG1}} == \texttt{SR}_{\texttt{SIG2}}$, which would be more natural, because the SMT solver will return satisfiable even if it finds only one solution for which the formula holds. In contrast, if the SMT solver returns unsatisfiable for the formula representing inequalities, then the two SRs cannot be unequal, and thus they are equal. Next, we present details for each signature equality check step.

First, *Heuristic Check* ① applies a set of pre-conditions on signature properties. This step aims to detect inequality between two signatures without invoking the SMT solver. First, the length of $\overrightarrow{\texttt{TT}}$s from both the signatures are checked for equality. The length of a $\overrightarrow{\texttt{TT}}$ indicates the number of distinct transformed streams observed by a query. Second, the number of keys in *tts* from both $\overrightarrow{\texttt{TT}}$s are checked. The number of keys in a *tt* indicates the number of attributes in output tuples. Third, ISQM checks the length of $\overrightarrow{\texttt{WIND}}$s from both signatures for equality. The length of a $\overrightarrow{\texttt{WIND}}$ indicates the number of windows observed until an operator. As a result, for two operators to be equal, they must process the same number of transformed streams, return tuples with the same number of attributes, and have the same number of windowed operators. If

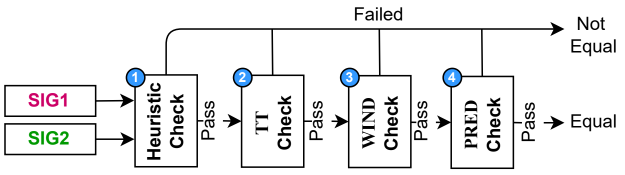


**Figure 4: Steps to compute equality between signatures.**

any of the above checks fail, ISQM marks the signatures and thus the operators as unequal. However, passing the heuristic check can still result in unequal signatures and thus operators. For example, two queries consuming tuples from same number of streams, applying same transformations, and producing tuples with same number of attributes can however consume data streams from completely different sources. Thus, ISQM next invokes an SMT solver to validate equivalence between the two signatures further.

Second, $\overrightarrow{\texttt{TT}}$ *check* ② identifies equivalence among tuple transformations captured by the signatures. Two equal operators must apply the same transformation on their input tuples. To this end, ISQM utilizes the SMT solver to check equality among $\overrightarrow{\texttt{TT}}$s from the two signatures. In particular, for each *tt* in one $\overrightarrow{\texttt{TT}}$, ISQM finds a matching *tt* in the other $\overrightarrow{\texttt{TT}}$. A *tt* represents a mapping between attributes in a tuple and the transformations applied to them. As the attribute order among the tuples must be the same, ISQM computes a formula for each pair of attributes representing inequality among their SRs. SMT solver returns unsatisfiable for the formula with matching SRs. Two *tts* are equal if, for each attribute pair, there exists a matching SR. $\overrightarrow{\texttt{TT}}$ check only validates if operators apply the same transformations on the tuples. However, operators applying the same transformations can still contain different windows in their upstream operator chains.

Third, $\overrightarrow{\texttt{WIND}}$ *check* ③ addresses these problems by checking equivalence across windows definitions captured by the two signatures. For equality, two operators must define windows with the same types and measures and perform the same operation (i.e., aggregation or join). To this end, ISQM checks equality between $\overrightarrow{\texttt{WIND}}$s from the two signatures. In particular, for each SR in one $\overrightarrow{\texttt{WIND}}$, ISQM finds a matching SR in the other $\overrightarrow{\texttt{WIND}}$. ISQM computes a formula for the inequality among the two SRs. The SMT solver returns unsatisfiable for the matching SRs. ISQM declares the two $\overrightarrow{\texttt{WIND}}$s equal if it finds matching SRs between them. However, operators applying the same transformation or containing the same windows can still have different predicates.

Lastly, PRED *check* ④ addresses this problem by identifying equivalence among predicates represented by the two signatures. Two equal operators must apply the same predicates on their input streams. To this end, ISQM computes a formula representing inequality among SRs from both PREDs. PREDs are equal if the SMT solver returns unsatisfiable for the formula. Overall, ISQM considers two signatures equal if all four checks pass.

### 6.2 Sharing Identification

The sharing identification phase utilizes the signature equality check described in Sec. 6.1 to identify sharing among queries. Signature equality between two operators indicates that the operators and their upstream operator chain (independent of operator order) are equal and thus can be merged to share data and computation. In contrast to previous work [47, 48], this phase enables our approach to identify both complete and partial sharing identification. Complete-sharing compares two queries together and only merges them if they are entirely equal. On the other hand, partial sharing identifies the maximum possible sharing among

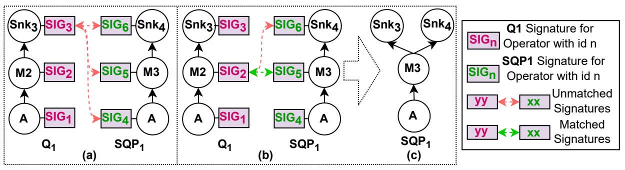


**Figure 5: Sharing identification between $\mathtt{Q}_1$ and $\mathtt{SQP}_1$.**

queries. To this end, this phase performs a two-step process: 1) selecting candidate SQPs and 2) sharing identification. First, it fetches candidate SQPs from the GQP based on interesting properties (e.g., sources) for sharing identification. Note that our approach can establish a trade-off between faster optimization time (by decreasing the number of candidate SQPs) and the amount of sharing it may find (by increasing the number of candidate SQPs). In our implementation, it computes an order-independent hash of the sources consumed by the queries and looks up the source index in the GQP to fetch candidate SQPs. Note that we use sources as an interesting property because queries must operate at least on the same stream sources to be equal. Second, after fetching candidate SQPs, the phase performs sharing identification for complete or partial sharing.

For complete sharing, in the second step, the sharing identification phase iterates over the collection of candidate SQPs and compares the signatures of each sink operators from both the new query plan and the iterated SQP. If signatures are equal, the matched operators and identifiers of the new query plan and SQP are sent to the next phase (see Sec. 3). Otherwise, the phase iterates to the next candidate SQP and repeats the signature comparison process until it finds a matching pair of sink operators. If it finds no match, it forwards the new query plan to the next phase for updating the GQP. We present more details on updating the GQP in Sec. 7. Note that, approaches such as query containment ([21, 22, 49]) can be applied on top of the GQP prepared by ISQM after sharing identification. In particular, containment-based approaches derive a representative query from a set of similar queries. In contrast, ISQM focuses on identifying and merging equal queries. Thus, ISQM is orthogonal and complementary to containment-based approaches, and we leave its integration for future work.

The first step returns only one candidate SQP that operates on the same source stream as the new query plan for partial sharing. In particular, the sharing identification phase iterates over the new query plan and the candidate SQP in breadth-first order. Iterating in the breadth-first order allows for identifying the maximum sharing opportunities as the phase compares the most downstream operators first for equality (in a top-down fashion). For each iterated operator in the new query plan, the phase iterates over the SQP until it finds an operator with a matching signature. It then forwards the pair of matched operators and identifiers of the new query plan and the SQP to the next phase and terminates. However, if the phase finds no sharing for the new query plan's operator, it repeats the second step for the next upstream operator in the new query plan. Overall, the source operators from the new query and SQPs have matching signatures as both process the same streams.

*Example.* Figure 5 shows how the sharing identification phase performs sharing identification between $\mathtt{Q}_1$ and $\mathtt{SQP}_1$. First, the signature $\mathtt{SIG}_3$ from $\mathtt{Q}_1$ is compared to the signatures from $\mathtt{SQP}_1$ in breadth-first order (i.e., $\mathtt{SIG}_6$,$\mathtt{SIG}_5$,$\mathtt{SIG}_4$). However, the phase finds no matching signatures due to semantic differences (Figure 5(a)). Next, the signature $\mathtt{SIG}_2$ from $\mathtt{Q}_1$ is compared to the signatures from $\mathtt{SQP}_1$ in breadth-first order. In this iteration, $\mathtt{SIG}_2$ matches to the signature $\mathtt{SIG}_5$ from $\mathtt{SQP}_1$ (Figure 5(b) green arrow). The phase then sends the pair of matched operators from the query and SQP to the global query plan update phase. We discuss more about the global query plan update phase in Sec 7.

## 6.3 Semantic based query Merging Variants

ISQM utilizes the combination of semantic information (see Sec. 4.2), a set of heuristics, and an SMT solver (see Sec. 6.2) to perform a comprehensive sharing identification. We refer to this approach as SM. However, SM incurs a high optimization time for two specific workloads during sharing identification.

First, for complete sharing identification in a query workload with many syntactically similar queries, SM can become suboptimal due to using a more extensive SMT solver for sharing identification. To mitigate this problem, we propose SM+ that improves the optimization time for syntactically similar queries during sharing identification. SM+ first applies a faster approach (e.g., hash-based signatures [26, 27]) for sharing identification and utilizes a more comprehensive SM approach only when the fast approach finds no sharing. We present a detailed evaluation of SM and SM+ in Sec. 8.2.

Second, for query workloads that only few operators (e.g., only sharing sources), the default top-down approach (from sink towards source operators) of SM might result in a long optimization time. In particular, for partial-sharing, SM starts with the sink operator of a new query plan and compares it with each operator in a candidate SQP for equality. In case SM finds no match, it repeats the process for the next upstream operator (see Sec. 6.2). However, this top-down approach (from sink towards source operators) of SM can result in a long optimization time when identifying sharing among queries that share few operators (e.g., only sharing sources). To address this problem, we modify SM to perform a bottom-up equality check, i.e., starting from the source to sink operators. We call this variant SM Bottom-Up or (SM-BU) for short. SM-BU improves the optimization time compared to SM by early identification of inequality and thus performing an early termination of sharing identification process. On the downside, SM-BU might identify fewer sharing opportunities compared to SM as it does not explore further downstream operators for equality. We present a detailed evaluation of SM and SM-BU in Sec. 8.3.

## 7 GLOBAL QUERY PLAN UPDATE PHASE

In this section, we discuss the *GQP update phase* in detail. We focus on identifying, representing, and managing sharing opportunities in a GQP for many continuously arriving and leaving stream queries. This phase modifies running GQP by incrementally adding matched/unmatched or removing running queries. In contrast to previous work [47, 48], this phase enables ISQM to manage sharing opportunities among running and newly arriving queries. ISQM redeploys updated GQP together for a batch of changes to prevent frequent redeployment. We keep redeployment out of the scope of this paper. However, ISQM can be configured to use more specialized solutions, such as incremental redeployment [27], for efficient query redeployment.

**Incrementally Adding a Matched Query**. The GQP update phase receives the sharing opportunities identified between a new query plan and an existing SQP. In particular, it receives a pair of matched operators and updates the SQP using a two-step process. First, this phase adds all downstream operators from the new query plan to the matched operator of SQP. If the matched operator from the new query plan is of type sink, the phase adds

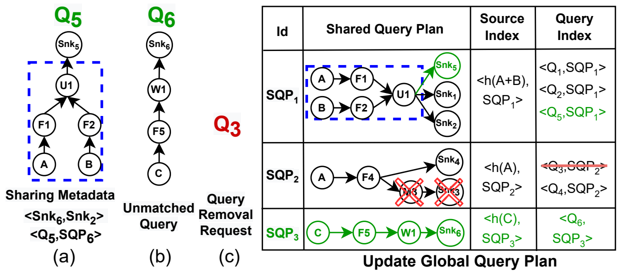


**Figure 6: Example query merging sequence.**

this operator to the upstream operator of the matched operator from SQP. This addition of operators allows the SQP to serve both already merged query plans and the new query plan. Second, this phase updates the query index by linking the ID of the new query plan to this SQP. This allows efficient removal of the query later.

*Example.* Figure 6 shows the addition of query plan $\mathtt{Q}_5$ to $\mathtt{SQP}_1$ in the GQP. The sink operator $\mathtt{Snk}_5$ of $\mathtt{Q}_5$ matches with the sink operator $\mathtt{Snk}_2$ of $\mathtt{SQP}_1$ (shown in blue). As both operators are of type sink, first the GQP update phase adds $\mathtt{Snk}_5$ to the upstream operator `U1` of $\mathtt{Snk}_2$. Second, the phase updates the query index by adding an entry mapping $\mathtt{Q}_5$ to $\mathtt{SQP}_1$.

**Incrementally Adding an Unmatched Query.** The sharing identification phase may find no sharing opportunity for newly arriving queries. As a result, the GQP update phase would receive the newly arrived query plans instead. In this case, this phase first computes a new SQP with all operators of the received query plans. Second, it updates the query and source indexes with information about the new query plan and the new SQP.

*Example.* In the bottom of Figure 6, we show a new $\mathtt{SQP}_3$ that replicates the received query plan $\mathtt{Q}_6$. In addition, this phase updates the query index by mapping $\mathtt{Q}_6$ to $\mathtt{SQP}_3$, and the source index by mapping the hash value for source name $C$ to $\mathtt{SQP}_3$.

**Incrementally Removing a Query.** The GQP update phase removes a query in four steps. First, it looks up the query index and fetches the SQP serving the target query. Second, it looks up an internal data structure to fetch the sink operator for the target query. Third, it removes the sink and the attached upstream operators that exclusively serve the target query, i.e., operators which are not shared with other queries in the SQP. In particular, this phase terminates the operator removal process when it encounters an operator that is shared with another downstream operator. This termination prevents removing operators which are serving other running queries. Finally, it updates the query indexes by removing the entries for the removed query.

*Example.* Figure 6 shows in red the removal of a query $\mathtt{Q}_3$ from the GQP. This phase updates $\mathtt{SQP}_2$ by removing the Snk operators$_3$, P1, and M3, which exclusively serve $\mathtt{Q}_3$. This phase retains `F4` and its upstream operators as they serve the running query plan $\mathtt{Q}_4$. Finally, this phase updates the query index by removing entries for the query $\mathtt{Q}_3$.

## 8 EVALUATION

In this section, we analyze the performance of `ISQM` using different query sets. First, we describe overall experiment setup and baselines (Sec. 8.1). Next, we analyze the overhead and sharing efficiency of baseline approaches and compare them with `SM` and its variants (Sec. 8.2 and 8.3). Afterwards, we analyze the overall resource utilization incurred while running query plans generated by different approaches (Sec. 8.4). Finally, we discuss the takeaways and limitations (Sec. 8.5).

### 8.1 Experiment Setup

**Hardware and Software Setup**. We run our micro-experiments to analyze the amount of sharing identification and optimization time taken by different sharing identification approaches. To this end, we use a Linux server with an AMD EPYC 7742 CPU, 1TB of main memory, and configure all techniques to use only a single core on the machine. For `SM` we use Z3v4.8.12 as the SMT solver. In our experiments, we observe that the time budget of Z3 SMT solver is essential for both the sharing identification and optimization time. Setting it too low or too high can result in either early termination (low sharing identification) or longer evaluation time (high optimization time) for signature equality check. We use a time budget of 1 ms that leads to best results in our experiments. We implement `ISQM` and all our baselines in NebulaStream [44] to prevent performance differences due to underlying system.

We run macro-experiments to analyze the runtime implication of query plans generated by different approaches. To this end, we deploy NebulaStream on a hierarchical cluster of eight Linux servers with 2 Intel Xeon Silver 4216 CPU, 500 GB of main memory, and 100 Gbit Infiniband connection. We configure each NebulaStream worker to use only 4 threads.

**Evaluation Metrics**. We adopt two metrics to validate the efficency of different sharing identification techniques: *Sharing Efficiency* and *Optimization Time*. Sharing efficiency is the percentage of operators from new queries merged into the GQP. Optimization time indicates the overall time taken between the arrival of a query and its addition into the GQP.

**Queries.** We evaluate the scalability of `ISQM` by using various syntactically and structurally distinct stream queries. As there exists no common benchmark for these workloads, we analyzed queries from open-source benchmarks, such as YSB [7], StreamBench [37], and identified commonly used operators in stream queries. Based on that, we develop an open-source *Query Generator* [1] that allows generating synthetic queries with different characteristics.

Our query generator takes as input the number of queries to generate, a set of source schemas, and configurations defining the composition of similarities (complete or partial) among the generated queries. First, the generator randomly selects one or two input source schemas and computes a seed query using their attributes. The generated seed query consists of a combination of several unary and binary stream operators (see Sec. 4). Note that the generator guarantees that all seed queries are semantically distinct. Second, the generator uses the seed query, the input configurations, and a collection of rewrite rules to compute semantically equivalent queries. We detail the input configurations for each query set together with each experiment. We refer the reader to our open-source repository for details on various rewrite rules used in query generation.

**Baselines Techniques**. We evaluate the sharing identification approach of `ISQM` against four baselines representative state-of-the-art techniques. To this end, we extend various phases of `ISQM` (e.g., signature computation, sharing identification) to implement these baseline approaches. *Sharing Identification based on Structural Analysis* (`SA`). This approach detects sharing opportunities by analyzing the structure of query plans for similarities. To this end, it traverses the query plans from the source to the sink operators and identifies equality among them. Systems such as SQPR [24], SharedDB [14], QPipe [19] adopt this approach for sharing identification.

[1]https://github.com/nebulastream/nebulastream-query-generator

*Sharing Identification based on Hash-based Signatures* (HB). This approach performs sharing identification in two steps. First, it computes signatures for each operator by traversing a query plan from source to sink operators. The signature of an operator is calculated based on the hash of the operator's syntax and the hash of its downstream operators. Second, HB compares operator signatures among the query plans to identify sharing opportunities. Systems such as AStream [27], BIGSUBS [23], NiagaraCQ [6] utilize HB for sharing identification.

*Sharing Identification based on Improved Hash-based Signatures* (HB+). HB considers syntactic information to compute an operator's signature. This consideration leads to different signatures for queries with syntactically different but semantically same operators. For example, HB computes different signatures for two plans with Filter predicates (c*b<a) and (b*c<a) because of the different attribute order. We address this issue by implementing HB+. In particular, we apply the *Attribute-Sort* and *Binary-Operator-Sort* rewrite rules to update input query plans. The Attribute-Sort rule sorts alphabetically the attributes in a Map or a Filter expression while retaining their semantic meaning. For example, the rewrite rule transforms the predicates of both example Filter operators to (a>b*c). The Binary-Operator-Sort rule sorts the upstream operators of a binary operator alphabetically based on their source names. For example, the rule swaps the sources A and B in the Join operator (B Join A) to (A Join B). Applying these rewrite rules eliminates simple sources of missing sharing opportunities and makes this approach more practical. Other than that, HB+ performs the same two steps as the HB for sharing identification.

*No Sharing Identification* (NoShare). We additionally perform all our experiments by disabling sharing identification. Using NoShare, we show the overhead incurred by different sharing identification approaches.

## 8.2 Identifying Complete Equivalence

The following micro-experiments evaluate different sharing identification approaches to detect complete equivalence among the input query set. In particular, we analyze the performance of different sharing identification approaches for increasing number of semantically equal queries 8.2.1, distinct sources 8.2.2, semantically distinct queries 8.2.3, and syntactically similar queries 8.2.4. We repeat each experiment three times and report the average sharing efficiency and aggregated optimization time.

### 8.2.1 *Increasing Semantically Equal Queries.*

In this experiment, we analyze the performance of our and other baseline sharing identification approaches with an increasing number of syntactically distinct but semantically equal queries. Additionally, we analyze the effect of heuristics (see Figure. 4) in our approach.

**Workload.** We generate synthetic queries that operate on four different streams. Overall, the query set consists of 100 distinct query groups[3]. In this experiment, we increase the total number of queries from 2K to 12K with increments of 2K. As a result, the sharing opportunities within a query group increase.

**Result.** Figure 7 shows the sharing efficiency and the aggregated optimization time for different sharing identification approaches. In general, the sharing efficiency and optimization time increase with more queries. The baseline HB achieves the lowest sharing efficiency across all query sets. In contrast, HB+ achieves between 5.6x to 7.9x higher sharing efficiency in comparison to

[3]Queries are semantically equivalent but syntactically different within each distinct query group. Across groups, queries are both semantically and syntactically different.

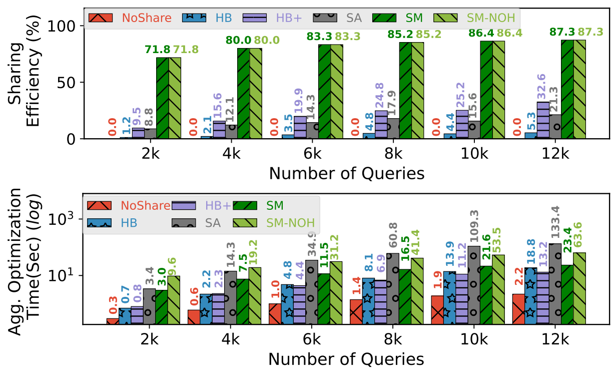


**Figure 7: Increasing semantically equivalent queries.**

HB. Additionally, HB+ shows a similar optimization time as HB for up to 4k queries, but outperforms HB for more queries. The baseline SA is slowest among all approaches and achieves a lower sharing efficiency than HB+ across all query sets. Our approach SM outperforms all baselines in terms of sharing efficiency as the number of queries increase. In particular, SM finds between 16.4x to 59.8x more sharing opportunities than the least efficient approach (HB) and between 2.6x to 7.5x more sharing opportunities than the most efficient baseline (HB+). In terms of optimization time, SM is between 1.1x to 5.7x faster than the slowest approach (SA) and between 1.7x to 4.2x slower than the fastest approaches (HB and HB+). Finally, without heuristics our approach (SM-NOH) takes between 2.5x to 3.2x longer optimization time.

**Discussion.** The increase in sharing efficiency and aggregated optimization time occurs across all approaches as more semantically equivalent queries are present in the query set, and thus the overall number of queries increase. HB and HB+ show the fastest optimization time among other approaches as they use a fast hash-based comparison for sharing identification. SA shows the longest optimization time as it uses an exhaustive graph isomorphism algorithm to identify sharing. As the number of equivalent queries increases, SA performs more work for sharing identification; thus, the overall optimization time also increases. This experiment confirms that SM identifies more sharing opportunities in comparison to other approaches because it captures the semantic information within a query and exploits it to identify sharing among queries with syntactic differences. Furthermore, the use of heuristics allows SM to efficiently detect inequalities among queries and thus enables a speeds-up of up to 3.2x comapred to SM-NOH. Overall, as SM utilizes an SMT solver to identify equality, it is up to 1.7x slower than the fastest approach (HB+) for 12K queries. However, for long-running stream queries, the cost of sharing identification amortizes over runtime and thus a higher query latency may pay off in terms of an overall improved resource utilization.

### 8.2.2 *Increasing Distinct Streams.*

This experiment analyzes the impact of increasing the number of distinct streams on the optimization time of sharing identification approaches. In particular, we analyze the effect of indexing SQPs based on the hash of stream names (Source Index) and pruning the candidates during sharing identification process.

**Workload.** We generate queries that operate on varying number of distinct sources. For each source, we generate 400 queries split in 10 distinct query groups. In this experiment, we increase the total number of distinct sources from 2 to 256. Note that with each new source, the total number of queries also increases.

**Result.** Figure 8 shows the aggregated optimization time for SM and HB+. Note that the reported optimization time is the aggregation of the time spent on identifying and merging sharing

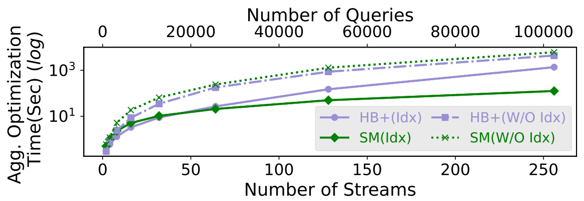


**Figure 8: Increasing number of distinct streams.**

opportunities for all queries in a query set (see top x-axis). For individual queries, the optimization time is in the order of milliseconds. We report numbers for only two sharing identification approaches as the overall trend in optimization time remains same across all approaches. We observe that the use of source index results in reduction of overall optimization time across both approaches as the number of sources (and queries) increase. However, the difference in optimization time is not profound for query sets with number of sources less than 8 with (`HB+(Idx)` & `SM(Idx)`) or without (`HB+(W/OIdx)` & `SM(W/OIdx)`) using source index. Overall, we observe that performing sharing identification using source index results in an improvement in optimization time between 1.8X and 5.7X for HB+ and between 2.2X and 48.8X for SM when the number of sources varies from 8 to 256. We do not report the numbers for sharing efficiency as the use of source index has no impact on it.

**Discussion.** In general, as the number of sources increases, the effect of using source index on sharing identification approaches becomes more prominent. For each new query, GQP looks up the source index to find candidate SQPs to identify sharing opportunities (see 6.2). For query sets with sources less than 8, the reduction in optimization time is not significant as the overall SQPs within the GQP remains low. Thus, lookup over all SQPs in a GQP for sharing identification do not add significant overhead. However, as the number of sources increases the overall candidate SQPs within the GQP also increases. Thus, using a source index allows pruning of candidates during sharing identification and results in significant reduction in overall optimization time across both the approaches.

### 8.2.3 *Increasing Distinct Queries.*

This experiment analyzes the impact of increasing the number of semantically distinct queries on sharing identification approaches. With the increase in the number of distinct queries, the overall sharing opportunities will reduce in a query set. Thus, we observe the effect of reducing sharing opportunities on sharing identification approaches.

**Workload.** We generate 4K queries that operate on four different streams. Overall, the query set consists of 100 distinct query groups for each stream. This experiment increases the percentage of semantically distinct queries in the query sets from 0 to 100%.

**Result.** Figure 9 shows the sharing efficiency and aggregated optimization time for all approaches. The sharing efficiency decreases with an increase in the percentage of semantically distinct queries across all approaches. Our approach SM outperforms all baselines in terms of sharing efficiency. SM finds between 33x and 107x more sharing opportunities compared to the least efficient baseline HB and between 5x and 13x more sharing opportunities than the most efficient baseline HB+. The aggregated optimization time remains constant as the percentage of distinct queries increases across all baselines. However, SM shows a linear increase in the optimization time. Till 40% distinct queries, SM shows between 1.4x to 2x faster optimization time compared to the slowest baseline approach SA. However, as the percentage of distinct queries increases by more than 40%, SM becomes up to 3.5x slower than SA.

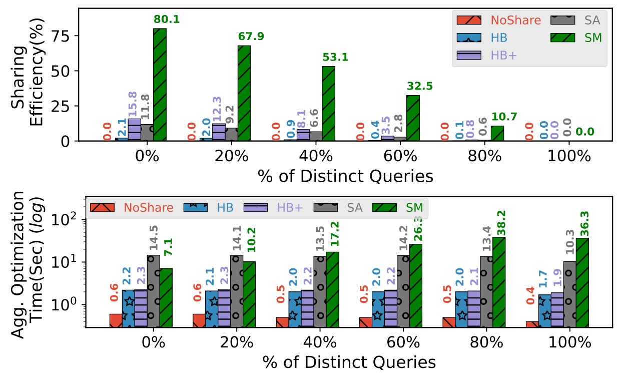


**Figure 9: Increasing number of distinct queries.**

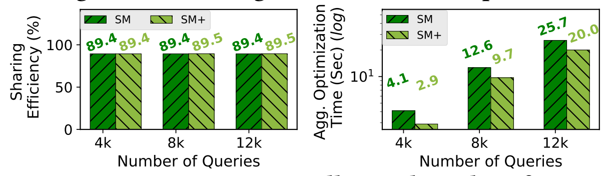


**Figure 10: Increasing syntactically equal number of queries.**

**Discussion.** The sharing efficiency decreases across all approaches with an increase in the percentage of distinct queries. Across baselines, HB+ outperforms all other in terms of sharing efficiency by employing additional rewrite rules to rewrite queries with only few syntactically differences into one similar query. SM achieves the highest sharing efficiency across all approaches as it exploits semantic relationships among queries for sharing identification. However, the aggregated optimization time of SM significantly increases as the number of distinct queries increases. This increase in SM's optimization time is because of the increase in the number of SQPs (due to unmatched queries) in the GQP. In contrast to the baselines HB and HB+, which use a fast hash comparison, or SA, which uses fail-fast graph isomorphism algorithm, SM uses a more costly SMT solver. In particular, SM performs more SMT calls to identify sharing for each incoming query as the number of SQPs increase. This results in longer aggregated optimization time for SM. However, we target large query workloads with hundrets of sources but thousands of queries in this paper and thus we assume queries induce at least a minimal amount of sharing opportunities. Furthermore, the optimization overhead is not significant when queries are operating over distinct streams (as shown in Sec. 8.2.2). Finally, the high cost of sharing identification will amortize over time for long-running queries.

### 8.2.4 *Increasing Syntactically Equal Queries.*

We observe that among the baseline approaches, HB+ achieves comparable sharing efficiency to SA and takes considerably less optimization time. Based on this observations, we prepare SM+ that combines faster HB+ with SM. In particular, SM+ first applies the fast HB+ IFF no sharing was found, it employs more comprehensive SM. In this experiment, we investigate the performance of SM and SM+ using query sets with syntactically equivalent queries.

**Workload.** We generate 4K, 8K, and 12K synthetic queries, uniformly distributed across four different streams. Unlike previous experiments, the query sets contain syntactically equal queries (before they were semantically equivalent).

**Result.** Figure 10 shows the sharing efficiency and optimization time for SM and SM+. We observe no change in sharing efficiency for both approaches but an increased optimization time with the increasing number of queries. In particular, SM+ is between 1.2x and 1.4x faster than SM.

**Discussion.** Both approaches achieve similar sharing efficiency with an increase in the number of queries for two reasons. First, the query sets consist of large volumes of syntactically equal queries that both approaches can detect. Second, SM+ uses SM as a backup check and thus achieves similar sharing efficiency as SM. However, SM+ achieves a faster optimization time in comparison to SM across all query sets. This improvement in optimization time is because the cheaper HB+ renders the use of more comprehensive SM unnecessary as queries are syntactically the same.

## 8.3 Identifying Partial Equivalence

In this micro-experiment, we investigate the performance of various baselines and variants of SM for partially equivalent queries.

**Workload.** We generate 4K queries that operate on four different streams. Overall, the query set consists of 100 distinct query groups for each stream. In this experiment, we increase the percentage of partial overlap from 20 to 100%. Thus, the overall sharing opportunities increase in the query set.

**Result.** Figure 11 shows the sharing efficiency and aggregated optimization time for all sharing identification approaches. HB achieves the least amount of sharing efficiency among all approaches and that do not conform to the trend of increasing sharing efficiency with increasing overlap. Both SM and SM-BU achieve higher sharing efficiency compared to the baseline approaches. However, SM-BU achieves between 3.4 to 8.8% less sharing efficiency in comparison to SM. Considering optimization time, HB and HB+ are the fastest sharing identification approaches, while SM shows the maximum optimization time across different approaches. However, we observe a decrease in optimization time for SM as overlap among queries increases. Finally in comparison to SM, SM-BU shows a speed up between 2.4x and 29.2x in optimization time.

**Discussion.** The sharing efficiency increases with the partial overlap among queries as all approaches discover more sharing opportunities. HB shows the least amount of sharing efficiency as it only relies on syntactic similarities for sharing identification and thus fails to find sharing in the presence of syntactic differences. SM achieves maximum sharing efficiency as apart from exploiting semantic information, it identifies sharing in a top-down fashion and performs an exhaustive search to find sharing. In contrast, SM-BU achieves less sharing efficiency than SM, despite exploiting semantic information, as it identifies sharing in a bottom-up fashion and terminates early if inequalities are detected.

The optimization time of HB and HB+ is fast as they employ cheaper hash-comparison for identifying sharing. In the case of inequality, these approaches terminate early as they look for sharing in a bottom-up fashion (similar to SM-BU). In contrast, SM takes maximum optimization time as it utilizes SMT solver and performs sharing identification in a top-down fashion. However, the optimization time for SM improves with an increase in partial overlaps as it identifies sharing early and terminates upon detecting equality among operators. In comparison to SM, SM-BU terminates early upon detecting inequality and thus achieves faster optimization time.

## 8.4 Resource Utilization

In this macro-experiment, we investigate the impact of running SQPs (generated by different sharing identification techniques) to determine the overall compute and network resource utilization.

**Setup.** We run 1K synthetic queries that operate on four different streams consisting of 10 distinct query groups. Overall, 50% of queries contain windowed joins and remaining 50% contains

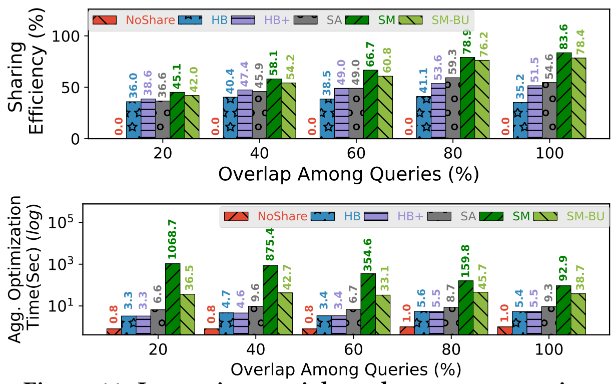


**Figure 11: Increasing partial overlaps among queries.**

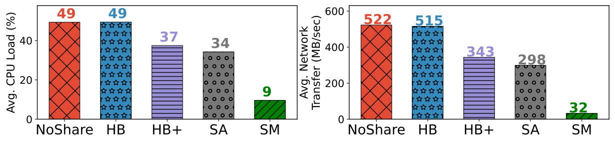


**Figure 12: Average compute and network resource consumption for 8 node cluster over a runtime of 120 sec.**

windowed aggregations and unions. We deploy the plans generated by each strategy on a NebulaStream cluster of 8 nodes (see Sec. 8.1). Each source produces data at a constant rate of ≈100 KB/sec. Note that, each of the 1K queries consume from at least two sources, resulting in an ingestion rate of over 100 MB/s.

**Result.** Figure 12 reports the average compute and network resource utilization of the overall cluster for the first 120 seconds of the experiment. SQPs generated by NoShare and HB consume similar compute and network resources. In contrast, SQPs produced by HB+ consumes less compute (24%) and network resources (34%) in comparison to NoShare. Furthermore, SQPs produced by SA consumes 30% less compute and 42% less network resources in comparison to NoShare. However, plans generated by SM consumes the least amount of resources with 5.4x less compute and 16x less network resources in comparison to NoShare.

**Discussion.** In this experiment, we show that the increased sharing efficiency of SM compared to other approaches has a real run-time impact on resource utilization. It demonstrates that the additional effort of finding sharing among queries pays off and thus is a valuable technique for modern SPEs. Important use cases for this are cloud environments where users pay based on resource consumption or to enable deployment of large workloads on non-elastic infrastructure (e.g. private data centers).

## 8.5 Takeaways

We evaluated ISQM and other baselines using query sets with different characteristics. We showed the runtime benefits of data and compute sharing for large numbers of long-running stream queries (see Figure 12). However, for a few short-running or non-overlapping queries, ISQM only adds overhead without identifying any sharing opportunities (see Figure 9). In the following, we recommend the best sharing identification techniques for different workloads.

We recommend our approach SM for syntactically different but semantically equivalent queries. SM captures semantic information in its query signatures and identifies hidden sharing opportunities resulting in a high sharing efficiency at the cost of increased optimization time (see Figure 7). However, for long-running queries, this increased optimization effort amortizes over time.

We recommend our variant SM+ (a combination of HB+ and SM) for a mix of syntactically similar and distinct queries. In particular, SM+ leverages fast HB+ to reduce optimization time while utilizing SM to identify sharing even in the presence of syntactic differences (see Figure 10). However, for partially overlapping queries, we recommend using our variant SM-BU that uses a fail-fast approach to improve the optimization time at the cost of reduced sharing efficiency (see Figure 11).

Overall, across all techniques, we observe a positive impact of using interesting properties for pruning the search space when performing sharing identification. In particular, we use GQP's source index to prune the candidate SQPs that are considered for sharing identification (see Figure 8). This reduction in the search space makes our approach practical for sharing identification among large numbers of queries.

## 9 RELATED WORK

**Signature-based Approaches**. Signature-based approaches compute signatures based on the query syntax and compare them with each other to identify sharing opportunities among queries. NiagaraCQ [6] uses signatures to group shared selection or join queries for thousands of batch queries. CSE [46] computes groups of shared queries using signatures, a set of heuristics, and a greedy algorithm. However, CSE scales only for tens of queries. BIGSUB [23] uses a mix of signatures and integer linear programming to group together thousands of shared sub-expressions. These three approaches target sharing identification among structurally similar batch queries. In contrast, ISQM focuses on sharing identification among structurally distinct stream queries. Karimov et al. [26, 27] use signatures for identification of sharing among thousands of ad-hoc stream queries. They utilize hash-based signatures to group together syntactically equivalent stream queries. In contrast, ISQM utilizes semantic information to identify sharing across thousands of syntactically different stream queries.

**Structural Comparison**. Another line of research proposes structural comparisons to identify sharing among queries. QPipe [19] compares query plans to maximize data and work sharing across concurrent batch queries. SQPR identifies equivalence among hundreds of stream queries by performing graph-isomorphism for efficient placement [24]. Chaturvedi et al. [4] propose an algorithm that computes a merged query plan by identifying the intersection of reusable tasks and streams. These approaches are designed for both batch and stream queries and can identify partial and complete sharing only among structurally and syntactically similar query plans. In contrast, ISQM utilizes semantic information to detect sharing even in the presence of structural and syntactic differences and scales for a large number of queries.

**Query Containment**. Approaches using Query Containment group a set of queries together and compute a representative query. The output from the representative query requires further splitting and filtering to serve individual queries. Zhou et al. utilize a set of heuristic rules to compute a representative query for Select, Project, Join, and Aggregate stream queries [49]. Hong et al. perform a more fine-grained exploration of operator-level containment relationships for batch and stream queries [21]. These approaches consider queries with syntactically different predicates or windows for sharing identification. However, they only group queries with structural similarities (same operator order) and require frequent re-optimization of representative queries as new queries arrive or old queries are removed. In contrast, ISQM considers semantically equivalent stream queries with syntactic and structural differences. Thus, ISQM can act as a pre-processing step when applying query containment for queries with syntactic and structurally differences. This allows ISQM to compliment query containment and enable a greater sharing identification.

**Semantic Analysis**. Approaches using semantic analysis check the semantic similarities among queries for sharing identification. Chu et al. propose COSETTE [9] and UDP [8] that apply rewrite rules to transform queries into algebraic expressions. However, these approaches do not scale as the application of rewrite rules is compute-intensive and does not support some widely used SQL features [47]. Zhou et al. propose EQUITAS [47] and SPES [48] to overcome these limitations using an SMT solver. However, these approaches are primarily designed for batch queries and can not be naively adopted for stream queries. In gernal, ISQM also uses an SMT solver but has several important differences in contrast to [47, 48]. First, ISQM supports sharing identification among stream queries by capturing window semantics in the signatures. Second, ISQM uses a GQP to represent, manage, and identify sharing opportunities among newly arriving and already running queries. Third, ISQM partitions a GQP into logically grouped SQPs and uses index over SQPs (source index) to prune the search space during sharing identification. Fourth, ISQM utilizes heuristics for speeding up the sharing identification optimization time. Overall, ISQM presents an end-to-end scalable solution for identifying, representing, and maintaining sharing opportunities for thousands of stream queries.

**General Approaches**. Madden et al. (CACQ) [28] and Chandrasekaran et al. (PSoup) [3] present approaches that allow cross-query work sharing for hundreds of queries. Both CACQ and PSoup use predicate indexing to group shared expressions. However, these approaches work only for structurally similar batch queries. In contrast, ISQM identifies sharing among stream queries even in the presence of syntactic and structural differences. Giannikis et al. present a combination of structural analysis and a branch-and-bound algorithm to identify partial or complete sharing opportunities among hundreds of batch queries [15]. In contrast, ISQM identifies partial and complete sharing among thousands of stream queries. Candea et al. present CJoin that exploits sharing opportunities among structurally similar data warehouse queries joining different dimensions and fact tables [2]. In contrast, ISQM utilizes semantic information to identify sharing opportunities among thousands of new and already running queries.

## 10 CONCLUSION

In this paper, we presented ISQM, an end-to-end solution for merging thousands of streaming queries. ISQM computes signatures that capture semantic information from a query. It utilizes these signatures to identify partial and complete sharing among syntactically distinct queries. Additionally, ISQM allows an efficient representation of identified sharing opportunities using shared query plans. This representation allows ISQM to exploit sharing among thousands of newly arriving and already running queries. Our evaluation showed that ISQM achieves a higher sharing efficiency and shows significant resource saving in comparison to all state-of-the-art baseline approaches. Additionally, we proposed two variants that outperform ISQM for specific workloads.

**Acknowledgement.** This work was funded by European Union's H2020 research and innovation programme - ELEGANT (No. 957286) and the German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and 01IS18037A). We thank Dr. Kaustubh Beedkar, Dimitrios Giouroukis, Philipp Grulich, and Dr. Varun Pandey for their valuable discussions and suggestions.

## REFERENCES

[1] Nikolaj Bjørner and Leonardo de Moura. 2014. Applications of SMT solvers to program verification. *Notes for the Summer School on Formal Techniques* (2014).

[2] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proc. VLDB Endow.* 2, 1 (2009), 277–288. https://doi.org/10.14778/1687627.1687659

[3] Sirish Chandrasekaran and Michael J. Franklin. 2003. PSoup: a system for streaming queries over streaming data. *VLDB J.* 12, 2 (2003), 140–156. https://doi.org/10.1007/s00778-003-0096-y

[4] Shilpa Chaturvedi, Sahil Tyagi, and Yogesh Simmhan. 2017. Collaborative Reuse of Streaming Dataflows in IoT Applications. In *13th IEEE International Conference on e-Science, e-Science 2017, Auckland, New Zealand, October 24-27, 2017.* IEEE Computer Society, 403–412. https://doi.org/10.1109/eScience.2017.54

[5] Ankit Chaudhary, Steffen Zeuch, and Volker Markl. 2020. Governor: Operator Placement for a Unified Fog-Cloud Environment. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 631–634. https://doi.org/10.5441/002/edbt.2020.81

[6] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data.* 379–390.

[7] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[8] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (July 2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 337–340.

[11] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (sep 2011), 69–77. https://doi.org/10.1145/1995376.1995394

[12] Apache Flink. 2021. *Overview | Apache Flink.* Retrieved Novemebr 15th,2021) from https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/overview/

[13] FLIR. 2022. *Acyclica by FLIR | Teledyne FLIR.* Retrieved September 18, 2022 from https://www.flir.com/products/acyclica/?vertical=public%20safety&segment=solutions

[14] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *Proc. VLDB Endow.* 5, 6 (2012), 526–537. https://doi.org/10.14778/2168651.2168654

[15] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7, 6 (2014), 429–440. https://doi.org/10.14778/2732279.2732280

[16] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[17] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20).* Association for Computing Machinery, New York, NY, USA, 2487–2503. https://doi.org/10.1145/3318464.3389739

[18] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2022. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (feb 2022), 196–210. https://doi.org/10.14778/3489496.3489501

[19] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 383–394. https://doi.org/10.1145/1066157.1066201

[20] Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. 2021. ESPBench: The Enterprise Stream Processing Benchmark. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '21).* Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/3427921.3450242

[21] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan J. Demers. 2009. Rule-based multi-query optimization. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings (ACM International Conference Proceeding Series)*, Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold (Eds.), Vol. 360. ACM, 120–131. https://doi.org/10.1145/1516360.1516376

[22] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. 2006. The Containment Problem for <bi>Real</bi> Conjunctive Queries with Inequalities. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06).* Association for Computing Machinery, New York, NY, USA, 80–89. https://doi.org/10.1145/1142351.1142363

[23] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[24] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter R. Pietzuch. 2011. SQPR: Stream query planning with reuse. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 840–851. https://doi.org/10.1109/ICDE.2011.5767851

[25] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.* IEEE Computer Society, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[26] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: Ad-Hoc Stream Joins at Scale. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 435–448. https://doi.org/10.14778/3372716.3372718

[27] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data.* 607–622.

[28] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02).* Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/564691.564698

[29] Microsoft. 2019. *IoT Signals report: IoT's promise will be unlocked by addressing skills shortage, complexity and security - The Official Microsoft Blog.* Retrieved September 18th, 2022 from tinyurl.com/2g85kg94

[30] Taewoo Nam and Theresa A Pardo. 2011. Conceptualizing smart city with dimensions of technology, people, and institutions. In *Proceedings of the 12th annual international digital government research conference: digital government innovation in challenging times.* 282–291.

[31] AP News. 2016. *Truck rams into German Christmas market, killing 12 people | AP News.* Retrieved July 8th, 2022 from https://apnews.com/article/802641a0498f41f3992874934cba4852

[32] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD '92).* Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/130283.130294

[33] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 51–63.

[34] Reuters. 2016. *Commentary: Nice attack – the wider threat to France | Reuters.* Retrieved July 8th, 2022 from https://www.reuters.com/article/france-attack-commentary-idUSKCN0ZV07S

[35] Reuters. 2020. *A look at Alibaba's Singles' Day, the world's largest online retail event.* Retrieved Jan. 31st, 2021 from https://graphics.reuters.com/SINGLES-DAY-ALIBABA/0100B30E24T/index.html

[36] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[37] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoT-Bench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* 29 (11 2017). https://doi.org/10.1002/cpe.4257

[38] Data From Sky. 2022. *Real-time road traffic analysis - FLOW by DataFromSky.* Retrieved July 8th, 2022 from https://datafromsky.com/flow/

[39] Snowflake. 2021. *Set Operators — Snowflake Documentation.* Retrieved November 16, 2021 from https://docs.snowflake.com/en/sql-reference/operators-query.html#general-usage-notes

[40] Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 97–108. https://doi.org/10.5441/002/edbt.2019.10

[41] Allied Vision. 2022. *Machine vision and embedded vision cameras for infinite applications - Allied Vision.* Retrieved Juls 8th, 2022 from https://www.alliedvision.com/en/applications/

[42] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th {USENIX} Symposium on Networked Systems*

*Design and Implementation ({NSDI} 20)*. 449–462.

[43] Wentao Wu, Philip A Bernstein, Alex Raizman, and Christina Pavlopoulou. 2020. Cost-based Query Rewriting Techniques for Optimizing Aggregates Over Correlated Windows. *arXiv preprint arXiv:2008.12379* (2020).

[44] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform for Data and Application Management in the Internet of Things. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[45] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. 2001. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 425–436. https://doi.org/10.1145/375663.375722

[46] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 533–544. https://doi.org/10.1145/1247480.1247540

[47] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability modulo Theories. *Proc. VLDB Endow.* 12, 11 (July 2019), 1276–1288. https://doi.org/10.14778/3342263.3342267

[48] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2020. SPES: A Two-Stage Query Equivalence Verifier. *CoRR* abs/2004.00481 (2020). arXiv:2004.00481 https://arxiv.org/abs/2004.00481

[49] Yongluan Zhou, Ali Salehi, and Karl Aberer. 2009. Scalable delivery of stream query result. In *Proceedings of 35th International Conference on Very Large Data Bases (VLDB 2009)*. VLDB.