Semantic Stream Query Optimization Exploiting Dynamic Metadata

Luping Ding [1] , Karen Works [2] , Elke A. Rundensteiner [3]

[1]*Oracle Corporation*
Nashua, MA USA
[1]lisa.ding@oracle.com

[2,3]*Worcester Polytechnic Institute*
Worcester, MA USA
[2]kworks@cs.wpi.edu
[3]rundenst@cs.wpi.edu

***Abstract*— Data stream management systems (DSMS) processing long-running queries over large volumes of stream data must typically deliver time-critical responses. We propose the first *semantic query optimization (SQO)* approach that utilizes dynamic substream metadata at runtime to find a more efficient query plan than the one selected at compilation time. We identify four SQO techniques guaranteed to result in performance gains. Based on classic satisfiability theory we then design a lightweight query optimization algorithm that efficiently detects SQO opportunities at runtime. At the logical level, our algorithm instantiates multiple concurrent SQO plans, each processing different partially overlapping substreams. Our novel execution paradigm employs multi-modal operators to support the execution of these concurrent SQO logical plans in a single physical plan. This highly agile execution strategy reduces resource utilization while supporting lightweight adaptivity. Our extensive experimental study in the CAPE stream processing system using both synthetic and real data confirms that our optimization techniques significantly reduce query execution times, up to 60%, compared to the traditional approach.**

I. Introduction

A. Exploiting Metadata

Many DSMS systems process long-running continuous queries over large volumes of real-time data. As data in these applications is real-time, meta knowledge about cardinality and data value arrival patterns is typically not obtainable at the time when compile time query optimization decisions are made. Similarly, no pre-built index for fast data access is available to be exploited for query processing at that time. Therefore, traditional optimization strategies, which heavily rely on pre-built indices, become inapplicable. Yet such stream systems may face scalability issues when processing hundreds or more of concurrent queries, as often experienced by applications [13], [20].

However meta knowledge on data values may become available as streaming data is generated [3]. One example of such metadata is network packet information for web service requests. If a given destination IP is highly requested, a router could batch such requests together and concurrently send metadata about the IP destination along with the packets. Another example is information generated for environmental management systems. Consider a fire detection application in a high-rise business complex where environmental sensor data is first propagated to a router that processes tuples within a specified region, then to a gateway that processes tuples for the respective floor and finally to the fire detection application. Correspondingly, metadata regarding such data (e.g., declaring that the next 5000 tuples will be from *regionID=26*) can easily be provided along with the actual data with little overhead as demonstrated by our experiments (Section 7).

Metadata on data values could also be derived by the query system itself [18]. For example, query systems often employ a buffer to collect input data. Data in the buffer may be pre-processed for a variety of reasons, including to sort for correcting out-of-order arrivals [18] or to perform load shedding [20]. Such pre-processing could annotate the data with relevant metadata. While pre-processing may incur some minimal overhead, the metadata provided could thus effortlessly be exploited to reap potentially significant benefit for a large number of down stream queries and applications as demonstrated by our experiments.

Motivating example: Reconsider the fire detection application mentioned above. The incoming sensor stream data is clustered by *regionID*. A metadata punctation could be inserted in front of each cluster to indicate the attribute values satisfied by tuples in the cluster [21]. We henceforth refer to such metadata punctuations as a *herald* as it is a "messenger" indicating the properties that a particular group of incoming tuples following it satisfy.

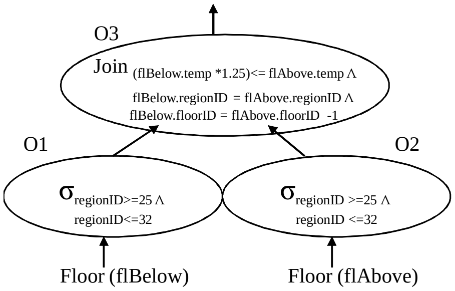

O3
Join (flBelow.temp *1.25)<= flAbove.temp Λ
flBelow.regionID = flAbove.regionID Λ
flBelow.floorID = flAbove.floorID -1
O1
O2
σ regionID>=25 Λ regionID<=32
σ regionID >=25 Λ regionID <=32
Floor (flBelow)
Floor (flAbove)

Fig. 1. Query Plan for Example Query Q1.

Consider emergency responders monitoring the progress of a fire by comparing the temperature of regions on fire on one floor to the same regions on the floor below. Query Q1, defined on a stream with schema *<floorID, regionID, sensorID, time, temp>*, reports region floor combinations for all regions with regionIDs within the range [25, 32] where the temperature observed on the upper floor (i.e., flAbove) is at least 25% more than the temperature observed at the same region on the next floor down (i.e., flBelow). The *Q1* query plan consists of 2 Select and 1 Join operator (Figure I-A).

Example Query Q1:

```
SELECT   flBelow.regionID, flAbove.floorID
FROM     FloorData flBelow, FloorData flAbove
WHERE    flBelow.regionID ≥ 25 and flBelow.regionID ≤ 32
and      flAbove.regionID ≥ 25 and flAbove.regionID ≤ 32
and      flBelow.regionID = flAbove.regionID
and      flBelow.floorID = flAbove.floorID - 1
and      (flBelow.temp * 1.25) ≤ flAbove.temp
```

Assume a herald *<regionID = [25, 32], time =* $*$*, temp =* $*$*; count, 5000>* is received, which indicates the next 5000 tuples all satisfy the condition *regionID* $\in$ *[25, 32]*. These 5000 tuples could bypass the evaluation of the region range predicate (i.e., operator O1) as they are guaranteed to satisfy it. Now assume that before reporting data, each router partitions the sensor data into partitions by temperature. If a herald *<regionID = [25, 28], time =* $*$*, temp = (*$-\infty$*, 100.0]; count, 5000>* is received, the data conforming to this herald can also bypass operator O1. Better yet, at operator O3, these tuples don't need to join with any tuples originating from the floor above that belong to the partitions *temp<125* as they are guaranteed not to produce any results. In short, the processing of these partition pairs could use a much more efficient query plan (i.e., eliminate the join) than the one selected without considering heralds. Clearly, such herald-driven optimization can result in significant savings in query execution costs.

B. Challenges in Herald Driven Optimization

Several important observations regarding herald driven query optimization can be made. First, heralds are only available at *runtime*. Hence query optimization must be conducted at runtime. Second, heralds have *lifespans*, i.e., the properties described by a herald are only satisfied by a particular substream. Thus optimizations driven by a herald will be applicable for limited periods of time (i.e., particular substreams). Finally, heralds from multiple streams considered in collaboration may enable several distinct optimizations.

For example, the herald <regionID = [25, 32], time = $*$, temp = [0.0, 100.0]> from stream $flBelow$ allows the join computation to be skipped if combined with the herald <regionID = [25, 30], time = $*$, temp = [150.0, 200.0]> from $flAbove$. The join is satisfiable if combined with the herald <regionID = [25, 30], time = $*$, temp = [90.0, 125.0]> from $flAbove$. Hence multiple distinct herald-optimized query plans may be valid at a time with *partially overlapped scopes* (i.e., the applicable substreams of these plans may overlap).

These properties raise serious challenges in designing new query optimization and execution techniques for exploiting heralds. First, the query optimization algorithm employed to find the optimized herald-driven plans must be efficient so as to identify optimization opportunities at runtime. Second, the algorithm must be lightweight to minimize the runtime optimization overhead. Also, a new query execution paradigm is required to support the concurrent execution of multiple logical plans on overlapping substreams without duplication of data storage or of processing. This requires logical plans on substreams to adaptively phase in and out with negligible physical plan switching costs [6], [22].

C. State-of-the-Art in Stream Processing

Semantic query optimization (SQO) [5], [14], which utilizes integrity constraints known at compilation time for query optimization, has been studied in the context of many considered *static* database systems from relational [5], object-oriented [11], to XML databases [19]. Most recently some SQO work on *stream* database systems has been reported [9]. However, these techniques are all conducted at compilation time and include detection of empty answer set and join/select elimination & introduction. Furthermore, they all produce one single optimized plan. We clearly face a different problem. Namely, the metadata to be exploited each have different scopes of applicability (i.e., may be valid for a limited period of time) and are only available at runtime. This requires lightweight constraint reasoning techniques that incrementally react to runtime metadata changes.

Present work on runtime query optimization for streaming data [2], [10] focuses on the traditional query rewriting problem of reordering operators in a query plan using selectivity statistics. No dynamic semantic knowledge has been considered thus far. In summary, existing work on SQO and runtime stream query optimization have been separate efforts. We are the first to combine them to conduct herald-aware query optimization at runtime. Existing works on punctuations focus on operator-level optimization, namely, on tuning the execution logic of individual join or aggregate operators [7], [8], [16]. Our work instead targets the query plan level, i.e. to optimize the overall plan structure to minimize execution costs by for instance skipping some operators completely.

D. Our Contributions: Herald Driven SQO

Our contributions can be summarized as follows:

1. We are the first to explore plan level semantic stream query optimization that exploits dynamic metadata. In particular, we identify four herald enabled semantic query optimization opportunities that parallel the traditional database SQO techniques [5], [14] (Section III-A).

2. To minimize the optimization overhead, we develop an incremental constraint reasoning algorithm named $PredSAT$ based on classic satisfiability reasoning theory. PredSAT is guaranteed to efficiently identify all four herald driven optimization opportunities at runtime (Section III).

3. Multiple concurrent SQO plans may be enabled by heralds for processing different yet potentially overlapping stream partitions. We propose a versioned minimum range model for supporting multiple concurrent logical plans proposed by the PredSAT algorithm (Section IV).

4. To achieve multiple concurrent logical plans with a single physical plan, we propose a novel query execution paradigm employing multi-modal operators with runtime configuration logic. This paradigm eliminates any replication of operator states or inter-operator queues, guarantees instantaneous application of herald driven query optimization, and requires zero plan migration effort (Section V).
5. Our extensive experimental study using both synthetic and real data streams confirms that herald driven optimization techniques significantly reduce query execution times, up to 60% (Section VIII), as compared to the traditional approach.

II. Preliminaries

Herald Model: Heralds are metadata punctuations [21] interleaved within streaming data. Each herald describes constraints on the attribute values of a sequence of tuples immediately following it via *attribute patterns*. Each corresponds to an attribute in the stream schema, indicating a range in the domain of that attribute and a *timestamp* indicating the scope of validity of a herald, also called a *lifespan*. A herald has an explicitly specified duration, either bounded (the more realistic scenario in practical applications) or unbounded (specified as ∞). Using a specified duration is a reasonable approach because it is not common that constraints in a stream will remain constant for a large amount of time, much less for infinity. A duration can be count-based (i.e., a given number of tuples following the herald) or time-based (i.e., a time range starting from the herald's timestamp).

Within the punctuation model, 1) tuples and heralds in the same input stream are assumed to be received in time stamp order [21], 2) the constraints must be valid, (i.e., tuples arriving within a herald's lifespan must conform to it), and 3) heralds in the same stream are assumed to not have contradicting constraints. Thus for any input stream, logically one herald is valid at any time, namely, the constraint implied by the strictest herald overwrites those with looser constraints. When a new herald arrives on a stream while a prior constraint is still active, its constraints are compared to the constraints in the current herald. The current herald is updated to the stricter of the two constraints. If the lifespan of the new stricter herald extends longer than the current herald's lifespan, then the lifespan of the current herald is updated to cover the lifespan of the new stricter herald. We henceforth focus on the crux of the problem (i.e., the semantic query optimization) and assume a single valid herald per each stream.

A herald is denoted as

$$(< ptn_1, ptn_2, ..., ptn_n >; < lifetype, lifeval >)$$

where ptn_i $(1 \leq i \leq n)$ is an attribute pattern for the i^{th} attribute A_i. *Lifetype* is the type of the lifespan either *count* or *time* which represent the count-based or time-based lifespan respectively. *Lifeval* is the value of the lifespan. A herald indicates that data following it *will* match the attribute patterns. Below is an example substream with a herald (in bold) and subsequent tuples that conform to it. The herald indicates that for the next 300 tuples following it, the *regionID* will be within the range of [200, 400).

schema: regionID, date, temp
(<[200, 400), *, *>: count, 300) – *herald*
(<201, 2008-02-01, 72.7>) – *stream data*
(<202, 2008-02-01, 75.6>) – *stream data ...*

Our Targeted Query. Our work focuses on optimizing predicates as found in join and select expressions. Henceforth, we focus on example queries containing such predicates. However, our method would equally work on queries containing additional operators such as group by or aggregation as the processing of these additional operators would remain unaffected. Our conjunctive queries are specified as follows:

```
SELECT <select-list>
FROM   <list-of-streams>
WHERE  <where-conditions>
[WINDOW <window-specification>]
```

The *where-conditions* are of the form $p_1 \wedge p_2 \wedge ... \wedge p_n$ in which 1) $p_i = z_1 \theta z_2$ (1≤i≤n); 2) z_j (j = 1, 2) is either an attribute or a constant, but z_1 and z_2 cannot both be constants; 3) θ is either =, < or ≤. Each predicate p_i is henceforth called an inequality predicate [12], though θ could be "=".

III. Constraint Reasoning

A. Herald Driven Optimization Strategies

Now, we describe four herald enabled query optimization cases. All are guaranteed to always lead to query execution cost reductions. Therefore, no cost-based decision is needed on deciding whether to apply our proposed optimizations. In the expressions below, we denote the ranges signaled by a herald (i.e., the inequalities already satisfied by the data) by d. Query predicates are denoted by q.

Select data skipping (SDS): If the selection predicate cannot be satisfied by the input data, the entire conjunctive query cannot be satisfied by the data. If any unsatisfiable selection predicate is identified based on the currently valid heralds, the select operation can skip the corresponding data. Expression 1 is an unsatisfiable selection predicate.

$$(A > 1000 \wedge A < B)_q \wedge (A < 800)_d \models False \tag{1}$$

Join data skipping (JDS): Similarly, whenever an unsatisfiable join predicate is detected based on heralds, the join operation can skip the corresponding data. Expression 2 is a join query over input streams A and B.

$$(A > 800 \wedge A < B)_q \wedge (B < 800)_d \models False \tag{2}$$

Select elimination (SE): If a selection predicate is known to always evaluate to true over the input data that conforms to a herald, it is a redundant predicate regarding the data. Such data can directly pass through the part of the query that was to evaluate this redundant predicate. Expression 3 is a redundant selection predicate where $A > 1000$.

$$(A > 1000 \wedge A \leq B)_q \wedge (A > 1200)_d \models (A \leq B)_q \tag{3}$$

Join simplification (JS): Similar to select elimination, any redundant join predicate regarding a certain herald need not be evaluated on each tuple that conforms to this herald. Instead, a cartesian product replaces the join predicate to simply combine the tuples without first having to execute this redundant join predicate. Expression 4 is a redundant join predicate where $A < B$.

$$(A < 800 \wedge A < B)_q \wedge (B > 1000)_d \models True \tag{4}$$

B. Completeness of Query Optimization

Select/Join data skipping are due to *unsatisfiable* Select/Join predicates, while Select elimination and Join simplification are due to *satisfied* Select/Join predicates respectively.

	Evaluation Result		
	True	*False*	*Unknown*
Select	Select Elimination	Query Pause	Regular Eval.
Join	Join Simplification	Query Pause	Regular Eval.

TABLE I

QUERY OPTIMIZATION OPPORTUNITIES.

For each predicate in our targeted queries (Section II), the result from evaluating an input is either true (satisfied), false (unsatisfiable), or unknown (yet to be evaluated) (Table III-B). Therefore, the above four optimization cases compose a complete set of semantic query optimizations based on predicate satisfiability. Each of these optimizations, once identified, ensures performance gains. There is no need for a complex cost-based query optimizer. Instead, a lightweight mechanism for identifying these optimizations is employed.

Among the four optimization techniques, select/join data skipping and select elimination parallel the cases covered by existing SQO techniques, namely, detecting an empty answer set and predicate elimination respectively [5], [14]. Join simplification is the runtime version of join elimination (JE) [5], [14]. They both are based on the identification of a redundant join predicate. Join is composed of the functionality of Cartesian product and predicates. JE identifies the join predicates for which both the Cartesian product and the predicates are redundant. Since Cartesian product determines the schema of the intermediate results, it is logically unavoidable regardless of any metadata on attribute values such as heralds. Therefore, at runtime join elimination reduces down to join simplification, which concatenates tuples from the two inputs without evaluating any predicates.

The identification of the applicability of any of the four herald driven optimization strategies outlined above can be mapped to the classic satisfiability and implication problems denoted as SAT and IMP [12].

Definition 1: Satisfiability Problem (SAT): Given a conjunctive formula S composed of a set of inequality predicates, the SAT problem checks whether at least one assignment for S exists such that S evaluates to true under the assigned values. If yes, S is said to be *satisfiable*. Otherwise, S is said to be *unsatisfiable*, denoted as $S \models False$.

Definition 2: Implication Problem (IMP): Given two conjunctive formulae S and T, both composed of a set of inequality predicates, the IMP problem checks whether every assignment that satisfies S also satisfies T. If yes, S is said to *imply* T, denoted as $S \rightarrow T$.

For both integer and real domains, one of the most effective SAT/IMP reasoning algorithms proposed in the literature is the *real minimum range algorithm* [12] or $RMin$ algorithm. The RMin algorithm has $|S|$ time complexity for solving the satisfiability problem and $|S|^2 + |T|$ time complexity for the implication problem for our targeted queries (Section II). Here $|S|$ and $|T|$ denote the number of predicates in formulas S and T respectively. Since our work extends RMin to make it employable for runtime query optimization (Section III-C), we now review RMin. RMin utilizes the *inequality graph* defined below (Definition 3) for representing the set of predicates.

Definition 3: Inequality Graph: An inequality graph for a conjunctive inequality formula S, denoted as G_S = (V_S, E_S), is a directed graph. Each node X in V_S one-to-one corresponds to a distinct attribute X in S. Each directed edge from node X to node Y in E_S, labeled with $\otimes$ and denoted as (X, Y, $\otimes$), one-to-one corresponds to an inequality (X $\otimes$ Y) $\in$ S. The label $\otimes$ is either $<$ or $\leq$.

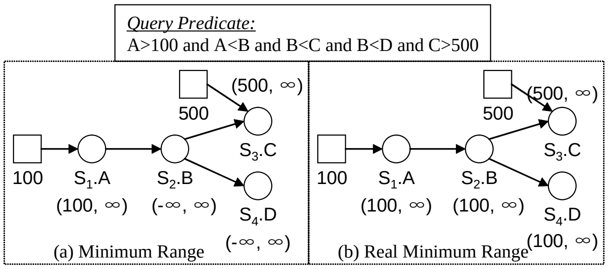


Fig. 2. Computing Real Minimum Ranges.

Figure 2(a) shows an example of the inequality graph for predicate *S1.A>100 and S1.A<S2.B and S2.B<S3.C and S2.B<S4.D and S3.C>500.* A circle denotes a variable and a square denotes a constant. The label of the edge, if not shown, is assumed to be $<$. Otherwise it is $\leq$.

We call a node Y a *parent node* of a node X (and X a *child* of Y) if X can reach Y via a directed edge. We denote the set of all parent nodes of a node X as *parents(X)* and the set of all children nodes of a node X as *children(X)*.

If two nodes X and Y in G_S are reachable via paths from each other, X=Y is implied by S via transitivity. All such variables and the edges among them are said to form a *strongly connected component*, or *SCC*. As an example, predicate *A≤B and B≤C and C≤A* corresponds to an SCC. By transitivity, it is equal to *A=B and B=C and C=A*. We use G_{S_c} to denote the *collapsed inequality graph* after collapsing each SCC in G_S into a single node. S_c denotes the *collapsed inequality formula* from S.

The RMin algorithm then works as follows. For an attribute X, let C^X_{up} = min(C_j) for all constants C_j such that X $\leq C_j \in S_c$. And let C^X_{low} = max(C_i) for all constants C_i such that X $\geq C_i \in S_c$. The closed range [C^X_{low}, C^X_{up}] is called the *closed minimum range* for X. These minimum ranges can be derived directly from the query. Figure 2(a) shows the minimum ranges for the attributes for a query with predicate *A>100 and A<B and B<C and B<D and C>500.*

The minimum range can be further refined to be the *real minimum range* [A^X_{low}, A^X_{up}] in which A^X_{low} and A^X_{up} denote the *real lower bound* and *real upper bound* of the attribute X respectively, computed as below. First, attributes in S_c are sorted in their topological order. Then attributes in S_c are selected one by one according to their topological order in S_c. For an attribute X, A_{low}^X = max(C_i, C_{low}^X) for all C_i such that C_i = $A_{low}^{X_i}$. Here X_i is X's child, if the edge from X_i to X is labeled with $\leq$; or C_i = $A_{low}^{X_i} + 1$ if the edge for X_i to X is labeled with $<$.

Next, we select attribute X one by one according to the inverse topological order of S_c. A_{up}^X = min(C_j, C_{up}^X) for all C_j such that C_j equal to $A_{up}^{X_j}$. Here X_j is X's parent, if the edge from X to X_j is labeled with $\leq$; or C_j = $A_{up}^{X_j}$-1 if the edge from X to X_j is labeled with $<$.

Figure 2 b shows the refinement of the real lower bounds of attributes S_2.B and S_4.D to be 100 instead of -∞ because these attributes are forced to be greater than S_1.A, whose real lower bound is 100.

Reasoning of the classic SAT and IMP problems using the real minimum ranges is based on Theorem 3.1. The proof can be found in [12].

Theorem 3.1: S is satisfiable iff 1) no SCC in G_S contains an edge labeled $<$ & 2) for each attribute X in S, $A_{low}^X \leq A_{up}^X$.

In addition, if S is satisfiable, S $\rightarrow$ T iff 1) for any (X $\leq$ Y) $\in$ T there exists a path from X to Y in G_{S_c}, or $A_{up}^X \leq A_{low}^Y$; 2) for any (X $<$ Y) $\in$ T there exists a path from X to Y in G_{S_c} with at least one edge of the path labeled with $<$, or $A_{up}^X < A_{low}^Y$; 3) for any (X $\leq$ C) $\in$ T, C $\geq$ A_{up}^X; and 4) for any (X $\geq$ C) $\in$ T, C $\leq$ A_{low}^X.
In the example in Figure 2, all predicates are satisfiable and no predicate can be implied by other predicates.

C. Proposed Incremental PredSAT Algorithm

We require near constant time performance for reasoning to enable herald driven optimization to occur at runtime. Thus we derive an efficient *incremental* reasoning algorithm for identifying the four herald driven SQO opportunities for the set of registered queries as new heralds arrive.

Herald driven query optimization can be abstracted to the following satisfiability and implication problems.

Definition 4: Herald Driven Satisfiability (Herald-SAT) and Implication (Herald-IMP) Problems: Given a set of inequality query predicates P_Q expressed by a query Q, and a set of inequalities P_D satisfied by input data D,
1. *Herald-SAT*: if $\bigwedge_{p_i \in (P_Q \cup P_D)} \models false$, select/join data skipping by p_i can be applied for data D;

2. *Herald-IMP_S*: for a selection predicate p_s in P_Q,
if $\bigwedge_{p_k \in (P_Q \cup P_D - p_s)} \rightarrow p_s$, select elimination can be applied to the predicate p_s;

3. *Herald-IMP_J*: for a join predicate p_j in P_Q,
if $\bigwedge_{p_k \in (P_Q \cup P_D - p_j)} \rightarrow p_j$, join simplification can be applied to the predicate p_j.
Here $\cup$ and $-$ denote the set-based union and difference operations respectively.

Heralds dynamically become valid at runtime. To identify possible optimization opportunities enabled by heralds would require the RMin algorithm to be invoked for each new herald received. The RMin algorithm has $|S|^2 + |T|$ time complexity (Section II) where $|S|$ and $|T|$ denote the number of predicates in formulas S and T. As the number of inequalities increases when more heralds are received, a significant reasoning overhead may be incurred. However, we observe that only a single pattern for a given attribute on a stream can be specified in each herald (Section II). Thus running the RMin algorithm over all inequalities triggers unnecessary reasoning. Instead, we design an incremental algorithm based on RMin called PredSAT (*for Predicate SATisfiability reasoning*). PredSAT limits the reasoning scope to only *relevant* inequalities, namely those that could enable any of the four optimization strategies (Section III-A) if combined with the new herald.

PredSAT algorithm: When a query is registered, the PredSAT algorithm constructs the inequality graph with real minimum ranges as induced by the query per RMin [12]. During query execution, PredSAT further refines the real minimum ranges based on the newly received herald. The refined minimum ranges are called *herald minimum ranges*. If a herald h with predicate $X \leq C$ is received, only the real upper bound of the node X and all the descendant nodes of X may possibly be affected (i.e., further tightened). Thus PredSAT starts from node X. If $A_{up}^X \leq C$, the algorithm stops. Otherwise, a refined upper bound C is computed for X. Then it proceeds to check X's children nodes in a breadth-first manner and terminates when an examined node has a real upper bound already < C. In the worst case where each node is connected to every other node the PredSat algorithm would have the same time complexity as RMin. In the average case, the time complexity is much smaller because only a small part of the query inequality graphs is traversed and in many cases after one local interval is adjusted within the graph the traversal terminates (Section VIII).
When computing herald minimum ranges, the PredSAT algorithm checks for optimization opportunities as follows:
1. *Unsatisfiable query.* P_Q is unsatisfiable if there exists a variable X in S_c, $A_{up}^X < A_{low}^X$.
2. *Redundant selection predicate.* For p_s: $X < C_q$,
$\bigwedge_{p_i \in (P_Q - \{p_s\}) \cup \{m\}} \rightarrow p_s$ if $\bigwedge_{P_Q - \{p_s\}}$ is satisfiable and $A_{up}^X > C$ (i.e., $C < C_q$). Therefore, p_s is redundant.
3. *Redundant join predicate.* For p_j: X $<$ Y,
$\bigwedge_{p_i \in (P_Q - \{p_j\}) \cup \{m\}} \rightarrow p_j$ if $P_Q - \{p_j\}$ is satisfiable and Y $\in$ parent(X)) and $A_{low}^Y > C$. Therefore, p_j is redundant
The proof is straightforward, hence eliminated for space.
Similarly, given a herald m: $X \geq C$, we determine
1. P_Q is unsatisfiable if $A_{up}^X < A_{low}^X$.
2. For p_s: $X > C_q$, $\bigwedge_{p_i \in (P_Q - \{p_s\}) \cup \{m\}} \rightarrow p_s$ if $P_Q - \{p_s\}$ is satisfiable and $A_{low}^X < C$ (i.e., $C_q < C$) .
3. For p_j: $X > Y$, $\bigwedge_{(P_Q - \{p_j\}) \cup \{m\}} \rightarrow p_j$ if $P_Q - \{p_j\}$ is satisfiable and Y $\in$ children(X)), $A_{up}^Y < C$.

Example: For query predicate *A>100 and A<B and B<C*

and B<D and C>500, the real minimum ranges are computed from the predicate (Figure III-Ca). When a herald indicating $S_2.B$ < *300* is received from S_2, the real upper bounds of attributes S_1.A and S_2.B are updated from ∞ to 300. The real upper bound of S_2.B (300) is now less than the real lower bound of S_3.C (500) (Figure III-Cb). Based on PredSAT algorithm, the join predicate $S_2.B < S_3.C$ now becomes redundant for processing substreams described by this herald and the output of evaluating selection on stream S_3. From the original query predicate, when a herald indicating $S_1.A$ < *50* is received, the real upper bound of attribute S_1.A is updated to be 50, which is lower than its real lower bound (100) (Figure III-Ce). This indicates that the predicate $S_1.A >$ *100* is unsatisfiable for the substream described by this herald. Similarly, Figures III-C c & f show the cases of redundant select predicate and unsatisfiable join predicates respectively based on the PredSAT algorithm.

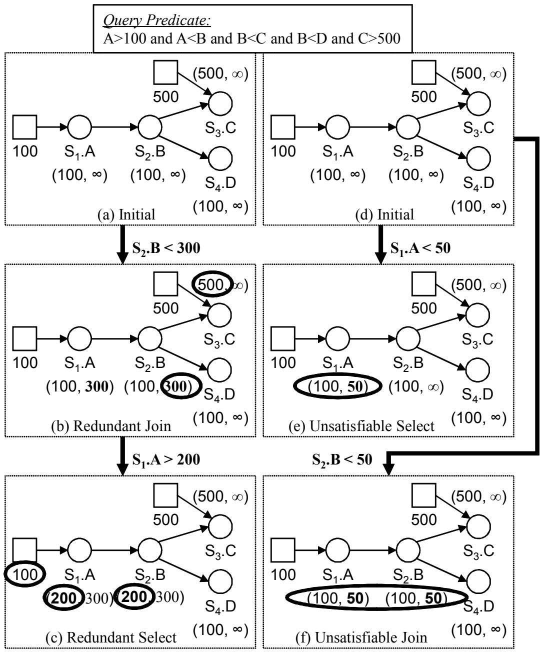


Fig. 3. Example of PredSAT Reasoning.

D. Mapping to a Logical SQO Plan

Once an optimization opportunity is identified for some substreams, an SQO plan can be generated to process these substreams. The remaining substreams would continue to be serviced by the default plan. The SQO plan generation follows the rules described in Equations 5 - 8 with ∪ representing *union all*. S_m, S_{m1} and S_{m2} represent the substreams described by heralds h, h1 and h2 respectively. σ_p, $\bowtie_p$ and $\times$ denote the select operation with predicate p, the join operation with predicate p, and the Cartesian product respectively.

$$Select\,elimination\,for\,S_h : \sigma_p(S_1 \cup S_h) \quad = \quad \sigma_p(S_1) \cup S_h \qquad (5)$$

$$Select\,data\,skipping\,for\,S_h : \sigma_p(S_1 \cup S_h) \quad = \quad \sigma_p(S_1) \qquad (6)$$

$$\begin{aligned} &Join\,simplification\,for\,(S_{h1}, S_{h2}) : && (7)\\ &(S_1 \cup S_{h1}) \bowtie_p (S_2 \cup S_{h2}) = \; S_1 \bowtie_p S_{h2} \cup S_2 \bowtie_p S_{h1} \\ &\qquad \cup S_1 \bowtie_p S_2 \cup S_{h1} \times S_{h2} \\ &Join\,data\,skipping\,for\,(S_{h1}, S_{h2}) : && (8)\\ &(S_1 \cup S_{h1}) \bowtie_p (S_2 \cup S_{h2}) = \; S_1 \bowtie_p S_{h2} \cup S_2 \bowtie_p S_{h1} \\ &\qquad \cup S_1 \bowtie_p S_2 \end{aligned}$$

SQO plan example: Once an optimization opportunity is identified for some substreams, an SQO plan can be generated to process these substreams. The remaining substreams would continue to be serviced by the default plan (Figure III-D). When the herald m with *A<500* is received, the join predicate $S_1.A < S_2.B$ is identified to be redundant regarding the substream described by m. Therefore, an SQO plan with the join operator replaced with a Cartesian product operator would logically need to be plugged in to process the substream described by m and the S_2 stream. The results of these two plans are merged to produce the complete final result.

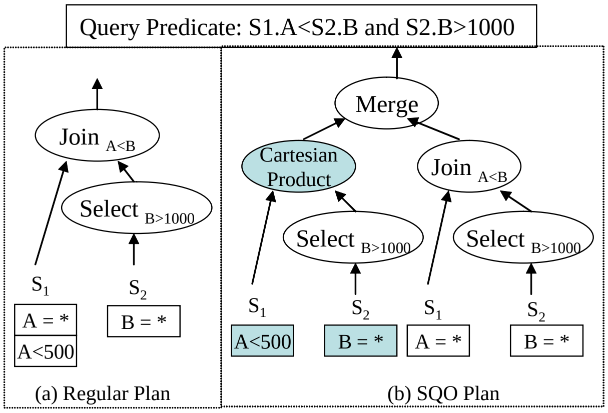


Fig. 4. Mapping Reasoning Result to SQO Plans.

IV. Multiple Logical Plans

During query execution, multiple heralds on the same stream attribute may be received over time. In addition, a single herald may enable multiple optimizations when combined with heralds from other streams. Thus we support overlapping validity scopes of multiple concurrent SQO plans.

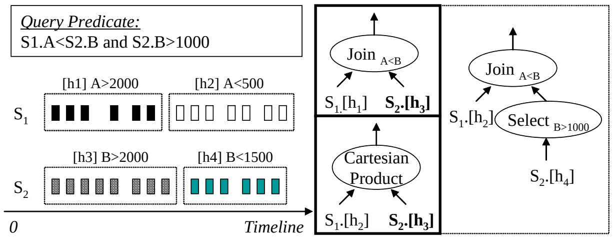


Fig. 5. Query Plans with Scopes.

Consider a query over two input streams S_1 and S_2 with predicates $S_1.A < S_2.B$ *and* $S_2.B > 1000$ (Figure IV). Assume that the data received from the input stream S_1 is described by two heralds in sequence: h_1: $A > 2000$ and h_2: $A < 500$. Also, the data received from stream S_2 is described by two heralds in sequence: h_3: $B > 2000$ and h_4: $B < 1500$.

The lifespans of these heralds are marked by the rectangles enclosing the substream of tuples directly below the herald predicate (bottom left of Figure IV). We denote the substreams that conform to these heralds as $S_1.[h_1]$, $S_1.[h_2]$, $S_2.[h_3]$ and $S_2.[h_4]$ respectively. Observe the following applicable optimizations to the given input substreams: select elimination (B>1000) of $S_2.[h_3]$ and join simplification (S_1.A<S_2.B) for ($S_1.[h_2]$, $S_2.[h_3]$), and join data skipping for ($S_1.[h_1]$, $S_2.[h_4]$). For substream pair ($S_1.[h_2]$, $S_2.[h_4]$), no herald driven optimization is applicable. Thus, for the input streams received so far, potentially four distinct query plans may need to be constructed to best serve each of these four cases. Such plans are called *SQO logical plans*. The *scope* of an SQO logical plan is defined as the set of substreams that need to be processed by the plan.

Herald h_1 contributes to the formation of two SQO logical plans, one solo and one in combination with another herald h_2. To capture all possible herald driven optimization opportunities based on all currently valid heralds, we now propose a *versioned real minimum range* concept. We call the real lower and upper bounds for each attribute X computed based on query predicates *query lower bound* (or Q^X_{low}) and *query upper bound* (or Q^X_{up}) respectively. Obviously, each attribute has a minimum range across a query referred to as the single query minimum range. The query upper and lower bounds of an attribute may be further refined by heralds received at runtime. As multiple heralds may be received for a single attribute, an attribute may be associated with multiple lower and upper bounds respectively (a.k.a. *herald lower* and *upper bounds*). The i^{th} herald lower and upper bounds are denoted as $A^X_{low,i}$ and $A^X_{up,i}$. The herald lower and upper bounds are maintained in two lists respectively associated with the attribute.

During query execution, each time a new herald is received, if either the herald lower or upper bound can be tightened then a new herald lower or upper bound is created and appended to the end of the corresponding list. Whenever such change occurs, optimization reasoning is triggered. During the reasoning, all herald bounds of the attributes visited are examined. For queries with windows, the PredSAT algorithm is applied and then the herald's lifespan is compared to the window constraint during query processing.

V. Runtime Execution

Given input streams containing heralds, multiple SQO logical plans may be concurrently applicable to different combinations of substreams. In addition, the scopes of the SQO plans may share common substreams. Thus, we cannot default to a traditional single-plan solution, which would need to employ an online plan migration technique [22] to continuously switch back and forth from the current plan to another plan in the middle of query execution. Instead, we must support the efficient execution of multiple query plans concurrently. However, to process tuples in multiple query plans may incur significant data duplication in operator input queues and states as different plans may share input and intermediate substreams. In view of this, we now propose a new query execution paradigm that tackles this challenge by supporting multiple concurrent logical query plans but physically corresponds to a single plan. The five key features of our proposed execution paradigm are:

1. *Data partitioning.* We partition data based on heralds. This allows different substreams to be served by the most suitable execution logic and then associate herald metadata with the respective stream subpartitions.
2. *Multi-modal operators.* We design query operators with configurable execution logic. Rather than using one algorithm for all incoming data, our query operators, guided by the query optimizer, apply customized algorithms to data from different stream partitions.
3. *Lightweight control table.* We design a control structure that enables customized execution logic for particular stream partitions by toggling a flag in the control table located in each query operator upon arrival of a herald-aware partition.
4. *Isolated operator tuning.* The configuration of an operator's logic is internal to the operator itself. Being localized, it does not affect the functioning of other operators nor the correctness of overall query processing.
5. *Partition propagation.* Each operator is equipped with the ability to propagate data partition information. This allows the configuration of a downstream operator without repartitioning.

Our *multi-modal operators* support the implementation of several logically distinct query functionalities. More precisely, each multi-mode operator realizes the physical processing of several different logical operators. For example our join operator may act as a theta-join, cartesian product, or no operator at all using the *evaluate*, *pass*, and *skip* modes respectively. Multiple logical query plans can thus be represented by one single plan composed of multi-modal operators. At runtime our query plan "adapts" the processing to react to both traditional and herald supported tuples realizing different logical operators and thus different logical plans.

Some significant advantages of our execution paradigm are:

1. It avoids data duplication by physically maintaining a single plan.
2. It avoids duplicate computations for tasks such as state insertion or purging or due to multiple logical plans working on overlapping input substreams.
3. It reduces system overhead by avoiding context switching among the otherwise much larger set of operators and even between different plans.

A. Multi-Modal Operators

To assure agility of operators, we equip our herald query operators with multiple execution modes configurable at runtime. That is, the operator processes every batch of data described by a herald in its most efficient manner as determined by the optimizer. This achieves multiple SQO logical plans within one *single* physical plan.

To configure its execution logic at runtime, each operator is equipped with a *control table* containing instructions on how to process herald partitions. The operator uses the herald associated with the partition to probe the control table and get the corresponding instruction. Based on the instruction, the operator applies the appropriate execution strategy to the current data partition. For instance, a select operator may either directly output the partition (select elimination), drop the partition (select data skipping) or evaluate the partition using regular predicate checking.

The control table is probed each time a new herald partition is received. Thus it needs to be probe-efficient. We implement the control table as a hash table with the partition ID as hash key. Thus instructions for a given partition are retrieved with a single lookup. The control table is updated at runtime by the *operator configurator component* of the herald driven semantic query optimizer. New entries are added into the control table as new herald-driven SQO opportunities are identified. To prevent the control table from growing in an unbounded fashion, existing entries are removed when the corresponding partitions have been processed.

B. Multi-Modal Select Operator

The select operator differentiates between three types of data partitions: 1) *Pass partition* where all tuples are guaranteed to satisfy the selection predicate; 2) *Skip partition* where all tuples are guaranteed to not satisfy the selection predicate; and 3) *Unknown partition* where it is unknown if any of its tuples will satisfy the selection predicate or not.

The select operator directly propagates any *pass* partition to its output stream (due to select elimination) and discards any *skip* partition (due to query pause) without evaluating any of their tuples. Tuples in the *unknown* partitions will be evaluated against the selection predicate via the regular select operator. This design enables one single operator to achieve three distinct query plans by applying three distinct logics to process its input data.

In the control table of the select operator, each hash entry contains a list of <PartitionID, ActionFlag> pairs. The action flag can be one of three values: 0 to pass, 1 to skip, and 2 to evaluate (for Pass, Skip, and Unknown partitions respectively).

During query execution, when an input partition is received, the select operator first checks whether it is an anonymous partition (i.e., with partition ID 0). If yes, then no herald is associated with this partition. Thus, the select operator evaluates tuples in this partition per the regular select operator. Otherwise, the partition ID is used to probe the control table. If a match is found, the corresponding action flag will be used to trigger the suitable execution logic to be applied to the tuples in the partition.

C. Multi-Modal Join Operator

The multi-modal join operator is associated with two control tables corresponding to its two input streams respectively. Similar to the control table of select, each control table for the join operator is hashed on the partition ID. Each hash entry contains a list of <LeftPID, RightPID, ActionFlag> triples indicating if the corresponding pair of partitions should be passed (per join simplification), skipped (per join skipping), or evaluated.

When a new partition p is received from the left input stream, the join operator first checks its partition ID. If it is an anonymous partition, the join operator processes tuples in this partition as a regular join (i.e., no optimization is done). Otherwise, the partition ID is used to probe the control table for the right input. If a match is found, the <LeftPID, RightPID, ActionFlag> triples in the list are enumerated. For each triple, the join logic indicated by $action$ is applied to the left-side partition with ID *LeftPID* and the right-side partition with ID *RightPID*. Processing of partitions received from the right input stream is similar.

Consider the example in Figure IV. If the partitions corresponding to heralds h_1, h_2, h_3 and h_4 have partition IDs 1, 2, 3 and 4 respectively. Then the control table of the select operator has one entry (3, Pass). The control table for the left input of the join operator (i.e. S_1) has two entries with keys being partition IDs 1 and 2 respectively. The partition ID 1 entry contains a list with one element (1, 4, DROP). The partition ID 2 entry contains a list with one element (2, 3, PASS). Correspondingly, the control table of the right input of the join operator, which is the output of the select operator, has two entries with keys being partition ID 3 and 4 respectively. The partition ID 3 entry contains a list with one element (3, 2, PASS). The partition ID 4 entry contains a list with one element (4, 1, DROP).

D. Partition ID Propagation

The herald driven data partitioning is initially conducted for source input streams. For the proposed semantic optimization to be applied to non-leaf operators as well, the partition IDs associated with source stream partitions need to be propagated through the query plan.

The propagation rules for the select operator are as follows. Each Pass partition is sent to the output stream of the select operator with its current partition ID. For each Unknown partition, if at least one tuple satisfies the selection predicate, a result partition is created with the current partition ID and will contain all tuples in the input partition that satisfy the selection predicate.

The join operator each time processes a new partition from one of its inputs, joining it with all existing partitions in the state of the other input. For each pair of partitions that may produce join results, the operator creates a result partition with the partition ID being the combination of the partition IDs of the two input partitions.

E. Herald Data Partitioning

Streams are partitioned based on heralds. A partition could be a source or an intermediate partition. Source partitions are obtained by partitioning source streams. There are two types of source partitions: 1) a *herald partition* contains tuples described by a single herald, and 2) a *anonymous partition* contains tuples not described by any herald. Intermediate partitions are produced as output of select or join operators.

An intermediate partition generated by a join may contain tuples from; 1) two source partitions, 2) a source partition and an intermediate partition, or 3) two intermediate partitions. Therefore, an intermediate partition can also be a herald partition, containing tuples described by n herald(s) (n≥1), or an anonymous partition.

Each source partition is assigned a stream-wise unique ID. The default partition ID 0 is reserved for any anonymous partition, while each new herald source partition is assigned the next available partition ID. The partition ID of an intermediate partition is the concatenation of the partition IDs of its component partitions.

VI. Herald-Aware Stream Engine

The framework of our herald aware stream processing engine is shown in Figure VI. The arrows represent communication between components. When a query is registered in the stream engine, the *static query optimizer* is invoked to conduct traditional query optimization without considering any heralds. Then the execution plan is sent to and executed by the *query execution engine*. The *runtime query optimizer* dynamically optimizes the query during execution. It is composed of the *statistics-based* and the *herald driven semantic query* optimizer. The statistics-based optimizer adjusts the query plan shape based on statistics about operator selectivities [10] gathered by the *statistics collector*.

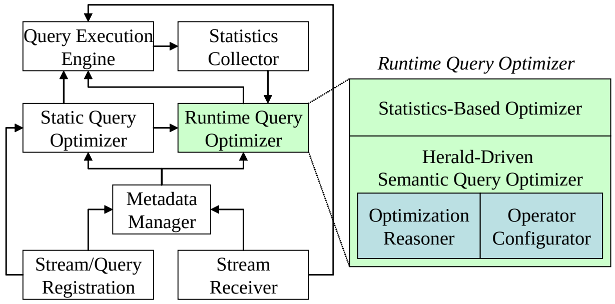


Fig. 6. Herald-Aware Stream Engine.

The *herald driven optimizer*, the focus of this work, is then continuously applied in the latest query plan as described in this paper. In particular, it consists of the *optimization reasoner* and *operator configurator*. Each time a herald is received, the *herald driven optimizer* is invoked. During each of its runs, the *reasoner* identifies new optimization opportunities and computes the SQO logical plans. Then the *operator configurator* configures the control table of the corresponding operators which allows the SQO logic plans to be realized.

Our experiments demonstrate that the *reasoner* has negligible overhead (Section VIII). Even if it were to ever lag behind the query execution, the correctness of query processing would not be affected. Rather it simply would cause the optimization of operator shortcutting to not be maximally exploited, i.e., not all possible optimizations may be applied.

The stream receiver feeds the tuples to the *query execution engine*, and forwards heralds to the *metadata manager* which maintains both integrity and runtime herald constraints.

VII. Herald Cost Analysis

We now analyze the estimated execution time of processing a given workload of tuples using heralds. Notations are in Table II. The execution time of a workload containing nt tuples processed using traditional query processing simply equals the regular processing costs to process the nt tuples ($nt * T_{Q_i}$). In contrast, the execution time of a workload containing nt tuples and nh heralds processed using heralds (WL_{Q_i}) equals the regular processing costs to process the nt tuples ($nt * T_{Q_i}$) plus the overhead to support the nh heralds ($nh * H_{oh}$) minus the cost eliminated due to some operators being skipped ($\sum_{i=1}^{no} ns_{op_i} * T_{op_i}$) (Equation 9).

Notation	*Meaning*
Q_i	a query
op_i	an operator
nh	number of heralds in a given workload
no	number of operators op_i in Q_i
ns_{op_i}	number of times operator op_i is skipped by a given workload
nt	number of tuples in a given workload
T_{Q_i}	est exec time of a tuple through Q_i using regular processing
WL_{Q_i}	est exec time of a given workload through Q_i using herald processing
T_{op_i}	avg per-tuple execution time of op_i
H_{oh}	estimated overhead to support a single herald

TABLE II

Notations.

$$WL_{Q_i} = (nt * T_{Q_i}) + (nh * H_{oh}) - (\sum_{i=1}^{no} ns_{op_i} * T_{op_i}) \tag{9}$$

The overhead to support heralds H_{oh} is the sum of the cost to execute the PredSat algorithm for each incoming herald (Section III-C) plus for each operator the cost to support multiple execution modes implemented via a simple hash lookup (i.e., constant lookup time). Our experiments (Section VIII) confirm that the overhead to support heralds is negligible compared to the skip savings, i.e., $(\sum_{i=1}^{no} ns_{op_i} * T_{op_i}) \geq (nh * H_{oh})$.

VIII. Experimental Study

A. Experimental Setup

Our experiments using both synthetic and real data streams compare the execution time of workloads using heralds (Herald) to traditional query processing (Regular). Each method is implemented in the CAPE stream engine [17]. The Windows XP test machine has a 2.66GHz Intel(R) Pentium 4 processor and 448MB RAM.

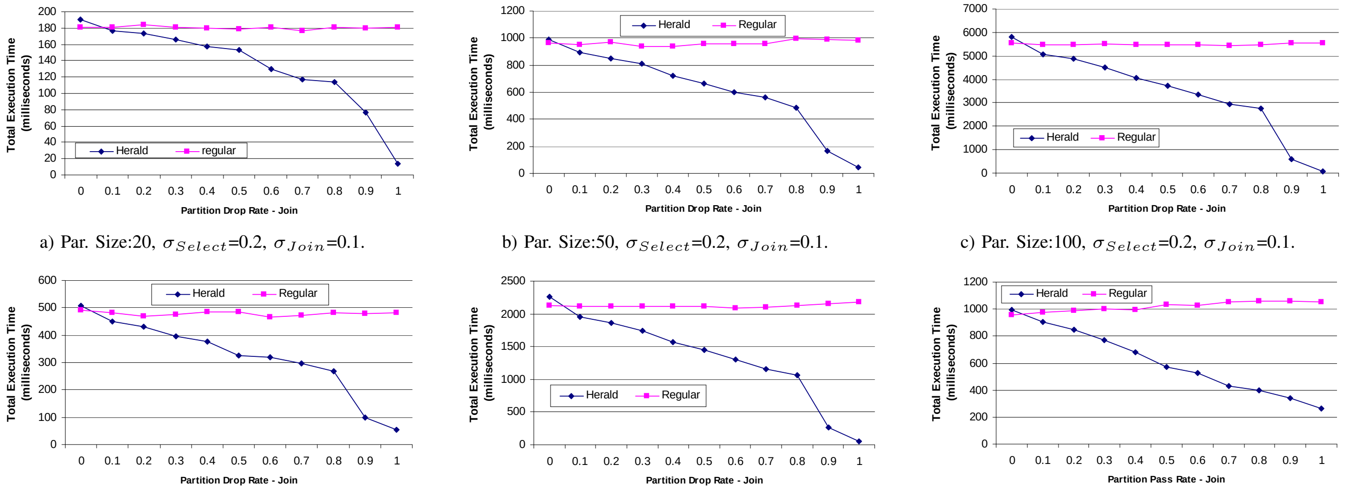
B. Evaluating Single Operator Optimization

First we use a synthetic data set to explore in controlled context the effect of different parameter settings on the performance. To focus on the cost saved by short cutting a single operator, we deploy the query in Figure IV.

Synthetic Data: We created a benchmark system to generate synthetic data streams that controls data distributions and arrival rates. Our experiments vary the following parameters.

1. *Average partition size* is the average number of tuples in each partition. The size follows uniform distribution.

2. *Selectivity* of the select operator is defined as $\frac{N_{out}}{N_{in}}$ with N_{out} and N_{in} (the total number of output and of input tuples



a) Par. Size:20, σ_{Select}=0.2, σ_{Join}=0.1.

b) Par. Size:50, σ_{Select}=0.2, σ_{Join}=0.1.

c) Par. Size:100, σ_{Select}=0.2, σ_{Join}=0.1.

d) Par. Size:50, σ_{Select}=0.1, σ_{Join}=0.1.

e) Par. Size:50, σ_{Select}=0.4, σ_{Join}=0.1.

f) Par. Size:50, σ_{Select}=0.2, σ_{Join}=0.1.

Fig. 7. Single Operator Optimization Experimental results.

respectively). Selectivity of join is $\frac{N_{out}}{N_1 * N_2}$ with N_{out}, N_1 and N_2 (the total number of output, of left and right input tuples respectively).

3. *Partition pass/drop rate* of the select operator is defined as $\frac{P_{pass}}{P_{in}}$ (or $\frac{P_{drop}}{P_{in}}$). Here P_{pass} (or P_{drop}) is the number of partitions that produce full (or no) results. P_{in} is the number of partitions. The partition pass (or drop) rate of the join operator is defined as $\frac{P_{pass}}{P_1 * P_2}$ (or $\frac{P_{drop}}{P_1 * P_2}$). P_{pass} (or P_{drop}) is the number of partitions pairs that produce full (or no) join results. P_1 and P_2 are the number of partitions from the left and right input respectively.

Partition drop rate: We first evaluate the effect of join data skipping. The partition drop rate is varied from 0 to 1 in increments of 0.1 to control the frequency of skips (Figure 7 b). The partition drop rate is the number of partitions that can be eliminated from processing (i.e., dropped). When the partition drop rate is low (i.e., close to 0), few partitions are dropped. While when the partition drop rate is high (i.e., close to 1), many if not all partitions are dropped. The average partition size is set to be 50 tuples. The selectivities of the select and the join operators are 0.2 and 0.1 respectively. As the partition drop rate increases, i.e., ns_i increases (Section VII), the herald execution time decreases (as HT_{Q_i} decreases). While no significant change in the regular approach's execution time is observed (i.e., $nt * T_{Q_i}$ remains constant) (Figure 7 b). When the drop rate reaches 0.9, the herald approach has more than 80% reduction in execution time compared to the regular approach. This result is promising because with just a small partition size (i.e., 50 tuples per partition) and low selectivity of the underlying select operator (i.e., when only 10% of its input data actually reaches the join), the herald approach already achieves significant performance gains.

Now, we evaluate select optimization. To test the optimization of select data skipping, we vary the partition drop rate. The average partition size is set to 50 tuples. The selectivity of the select and join operators are both set to 0.1. No performance gains were observed by the herald approach. We thus increase the average partition size to 200 tuples, and reduce the selectivity of both the select and the join operators to 0.01. This time we observe a reduction in execution time by the herald approach (Figure 8). Clearly, select data skipping for relatively small partition sizes and moderate select and join selectivities can only achieve modest performance gains because the cost of join is dominant.

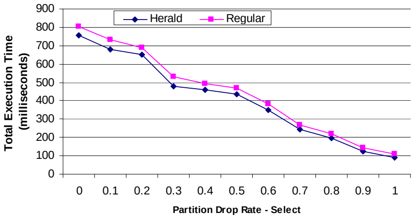
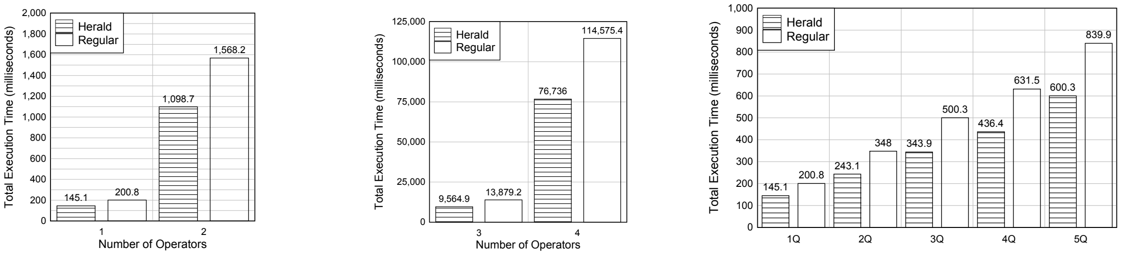


Fig. 8. Select Data Skipping Experimental results.

Partition sizes: Next, we investigate the role the average partition size plays in affecting the performance gains by studying scenarios with different partition sizes, namely, 20, 50, and 100 as depicted in Figures 7 a, b, and c respectively. All the other configurations remain the same as in the previous experiment. While the trends in all three experiments is similar, the gains achieved by the herald method increase as the partition size increases. This is due to the fact that the amortized optimization overhead is reduced by the increase in batch size (i.e., $nh * H_{oh}$ is reduced).

Operator selectivity: Next, we study the effect of selectivity of an operator. The selectivity of the underlying select operator determines the number of tuples to be processed by the subsequent join operator. We vary the selectivity of the select operator from 0.1, 0.2, to 0.4, average partition size is again 50 tuples, and the join selectivity is 0.1, as depicted in Figures 7 d, b, and e respectively. When the selectivity increases, the performance gains by the herald-driven SQO also increase (Figures 7 d, b, and e). Note here that each query to begin with will have different costs (see changes in y axis) due to the varying selectivity. This is because when the selectivity of the select operator increases, more data is processed by the join



a) Par. Size:50, σ_{Join}=0.1, Drop Rate=0.5. b) Par. Size:50, σ_{Join}=0.1, Drop Rate=0.5. c) Par. Size:50, σ_{Join}=0.1, Drop Rate=0.5.

Fig. 9. Multi-Query Workloads Experimental results.

operator (i.e., nt increases). Bigger performance gains can thus be achieved by employing herald-driven optimization because in essence the partition sizes arriving at the join remain larger (i.e., ns_i increases).

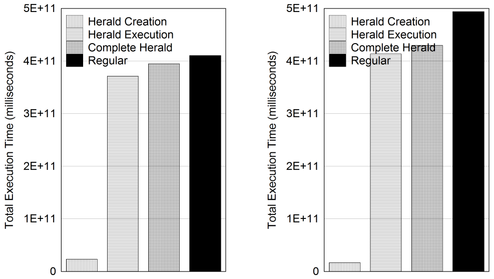
Pass rate: Now, we evaluate the impact of the pass rate. The average partition size is 50 tuples. The selectivities of the select and the join operators are 0.2 and 0.1 respectively. The partition pass rate is varied from 0 to 1 by 0.1. Similar to the drop rate, the herald approach significantly reduces the execution time (i.e., HT_{Q_i} decreases) as the partition pass rate increases (i.e., ns_i increases) (Figure 7 f).

C. Multi-Query Workloads

Next using the synthetic data we measure the costs saved by short cutting multiple operators when varying the query complexity and workload.

Query complexity: First, we vary the number of join operators in a single query. Each query executes a pipeline of join operators. The average partition size is 50. The partition pass/drop rate is set to 50%. Each join operator has a selectivity of approximately 10%. By varying the number of operators from 1 to 4 (Figures 9 a and b) herald outperforms the regular approach by 29.5% on average. The performance (i.e., how much more results the herald approach produced compared to the traditional approach) of a query containing a single join operator was 27.7%. For each join added to the query the cost saved slightly increases. This is as expected because as the number of joins in the query increases so does the number of opportunities for herald aware optimizations to take place (i.e., as no increases ns_i may increase). This experiment demonstrates that for complex queries the herald approach consistently outperforms the regular approach.

Number of queries: We now evaluate the overhead of workloads with different numbers of queries. Each query executes the join operator outlined in the above query complexity experiment. We vary the number of queries from 1 to 5 (Figure 9 c). On average the herald approach outperforms the regular approach by 29.7% indicating that the average savings per query within the workload are similar. The individual savings range from 27.7% to 31.2%. This experiment demonstrates that the gains made by heralds (i.e., skipping operators) are not affected by the number of queries in a system, rather it depends on the number and type of skip opportunities within the workload. Beyond that, running multiple queries using the herald approach has little overhead as the number of operators skipped by a query is unaffected by other queries.



a) Select- Drop Rate=0.75 b) Join- Drop Rate=0.75

Fig. 10. Real Data Set Results

D. Evaluating Optimization Using Real Data

Now we verify the effectiveness of the herald approach when applied to real data. We utilize a herald creation operator that pre-processes all incoming tuples and creates appropriate heralds. In these experiments we measure the cost required to create heralds and saved by short cutting operators.

Real Data Set: The real data was collected from 54 sensors deployed in the Intel Berkeley Research lab between February 28th and April 5th, 2004 (http://berkeley.intel-research.net/labdata/). We equally divided the sensors and group them group based upon their physical locations into 4 regions. In each experiment the herald creation operator clusters incoming sensor stream data by region.

Selection: To evaluate the optimization of select data skipping (Figure 10 a), we design a query for an environmental engineer responsible for locating all incoming sensor data from a single region. In this case data from three of the four regions is skipped. We observe that the cost to create heralds is indeed modest (about 5% of regular processing cost). A modest gain of 4% when comparing the complete herald approach (i.e., both the cost to create and execute heralds) to the regular approach can be observed.

Join: To evaluate the optimization of the join data skipping (Figure 10 b), we design a query for fire fighters monitoring the spread of a fire the reports the temperature of regions adjacent to a region on fire. In this case data from three of the four regions is skipped. We observe that the cost to create heralds was again very modest (about 3% of regular processing cost). In this case, the complete herald approach out performs the regular approach by 13%.

E. Summary of Experimental Findings

We summarize key findings here. First, we observed significant performance gains by using our herald-driven optimization when partition drop/pass rates are medium or high. Also, the performance gains of the herald approach increase as the partition drop/pass rate increases, as the selectivities of the underlying operators increase, and as the partition size increases. All experiments include the actual optimization reasoning overhead which is thus shown to be negligible. In particular, the overhead does not significantly increase due to query complexity and/or workload. In fact the overhead in all cases never diminished from the gains achieved by heralds.

IX. Related Work

Existing semantic query optimization (SQO) work employs schema knowledge or integrity constraints to perform compile time query optimization [9], [5], [14]. It has been extensively studied in traditional databases [5], [14] and more recently in *stream* database systems [9]. In the streaming context, [3] uses integrity constraints to optimize memory usage by purging operator states. We instead focus on utilizing dynamic metadata about attribute values to conduct efficient query optimization at run-time.

Existing work utilizing dynamic meta data (a.k.a. punctuations) focuses on the design of single query operators such as joins [7], [8] or the compile-time detection of "unsafe" queries with unbounded operator states [15]. [16] exploits punctuations to mark the sliding window boundary to handle disorder. Our focus is on plan level optimization enabling different stream partitions to be processed by logically distinct plans based on semantic knowledge. Also prior work only considers reordering of operators as in traditional databases whereas we instead partially or completely skip operators.

Much streaming database research has focused on runtime query optimization. [2], [10], [22] exploit runtime statistics on operator selectivities to adaptively reorder the operators in the query plan. Unlike our work, only a single logical plan is used during execution and no operators are skipped.

In Eddies [1], [6], individual tuples are adaptively routed through the operator network based on localized heuristics instead of using optimizer-generated query paths. Our work differs from Eddies in numerous significant aspects; 1) adaptation of Eddies is selectivity-driven while our adaptation is semantics-driven, and 2) we completely skip or pass certain query logic based on semantic knowledge. While Eddies only changes the order of operators as in traditional query optimization, but never skips any of them.

Similar to [4], we employ different plans for different data. But [4] adapts the operator execution orders for different data similar to traditional (syntactic) query optimization. In our work, to process a particular batch of data, some operators may be skipped while other operators may be executed in a more efficient way.

X. Conclusion

In this paper, we have proposed the first data stream SQO approach that exploits dynamic metadata at runtime. We designed four herald driven SQO techniques that once applied guarantee performance gains. A lightweight constraint reasoning algorithm based on classic satisfiability theory efficiently identifies SQO opportunities at runtime upon the receipt of heralds. To optimize resource usage in supporting multiple concurrent SQO plans with different yet overlapping scopes, a novel query execution paradigm employs multi-modal operators to achieve multiple logical plans with one single physical plan. Our experimental study using both synthetic and real data confirms that our herald driven optimization techniques significantly and consistently reduce query execution times compared to traditional DSMS approaches.

Acknowledgment: We thank D. Dougherty, M. Mani and peers in WPI database research group for useful inputs. This work is supported by the following grants: NSF 0917017, NSF CNS CRI 0551584, NSF 0414567, and GAANN.

References

[1] R. Avnur and et. al. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[2] S. Babu and et. al. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[3] S. Babu and et. al. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *TODS*, 29(3):545–580, Sep 2004.

[4] P. Bizarro and et. al. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.

[5] Q. Cheng and et. al. Implementation of two semantic query optimization techniques in db2 universal database. In *VLDB*, pages 687–698, 1999.

[6] A. Deshpande and et. al. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[7] L. Ding and et. al. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.

[8] L. Ding and E. A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[9] P. M. Fischer and et. al. Stream schema: Providing and exploiting static metadata for data stream processing. In *EDBT*, 2010.

[10] L. Golab and et. al. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[11] J. Grant and et. al. Semantic query optimization for object databases. In *ICDE*, pages 444–453, 1997.

[12] S. Guo and et. al. Solving satisfiability and implication problems in database systems. *TODS*, 21(2):270–293, 1996.

[13] M. A. Hammad and et. al. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[14] J. J. King. Quist: A system for semantic query optimization in relational databases. In *VLDB*, pages 510–517, 1981.

[15] H. Li and et. al. Safety guarantee of continuous join queries over punctuated data streams. In *VLDB*, pages 19–30, 2006.

[16] J. Li and et. al. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[17] E. A. Rundensteiner and et. al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[18] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[19] H. Su and et. al. Semantic query optimization for xquery over xml streams. In *VLDB*, pages 277–288, 2005.

[20] N. Tatbul and et. al. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[21] P. A. Tucker and et. al. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.

[22] Y. Zhu and et. al. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.