ABSTRACT
Groupby queries are prevalent in stream processing applications. We propose parameterized streaming groupby query (or PSGB query) to serve the needs for providing customized results based on widely-varying user requests. SPGB query is resource-efficient by assuming the pull execution model. In targeted applications, fast arriving streaming data and selective user requests can be observed. This raises the need for an indexing strategy that can effectively organize data in the quickly evolving groupby state to support heavy and fluctuating query workloads.

In this paper, we tackle this problem by proposing an efficient yet lightweight index solution named IMP index for SPGB groupby operator. We propose the EPrune algorithm that is guaranteed to find the optimal IMP index configuration for a given query workload. To support frequent index tuning required for coping with dynamic stream environments, efficiency of the index selection may become more important than guaranteed optimality. We thus design a greedy index selection algorithm named RGreedy and equip it with three alternative heuristics - OWL, RCL and Hybrid. Our experiments conducted in a real stream processing system show that RGreedy algorithm with the Hybrid heuristic finds optimal IMP configuration in a variety of test cases with xx% confidence interval. And while EPrune takes hours to finish, RGreedy always terminates within seconds.

1. INTRODUCTION

1.1 Groupby Queries in Stream Applications
Groupby queries with aggregate functions are extensively used in data stream applications to provide statistical summaries for monitoring and real-time analysis purposes.

Network monitoring applications run groupby queries over network packet data to monitor network traffic or to measure network performance [20, 23]. For example, Query 1 in Figure 1 monitors the total traffic of source-destination pairs in the past 20 minutes over a backbone link B.

Online transaction systems conduct groupby queries over transaction records to provide real-time recommendations to users. Query 2 (Figure 1) in online auctions [20, 21] computes the number of bids placed in the past 24 hours grouped by auction category, buyer's state and occupation.

Groupby queries are also extensively used in real-time business intelligence (BI) applications [10] where data arrives in a data warehouse or OLAP system via a trickle feed, continuously or every few minutes.

Most existing work in stream processing assumes the push execution model [4, 6, 12, 17, 22, 23], i.e., the query actively delivers the streaming result as time goes by, either when a new input tuple is received (sliding window) [5] or when a certain time period has elapsed (hopping window) [7].

However, we have observed that the push model can be fairly inappropriate for streaming groupby processing. First, groupby queries as in Figure 1 often correspond to numerous data groups. Under the push model, huge volumes continuous updates on groupby results would thus be repeatedly produced. Second, a particular user may be interested in the aggregate results of a few instead of all groups. Different users may request results for distinct groups. For example, the auction system may use Query 2 to produce recent transaction statistics customized to each user's preference. Users may have widely-varying preferences. Therefore, to deliver large amounts of results that are mostly useless to individual users may prevent them from finding the desired information nugget. It may also overload system resources, and as a result, negatively affect the query response time and the quality of those answers that are really needed by users. Instead, users may prefer to receive high-quality responses in real-time whenever they request them.

We also note that the push model requires that the query is statically specified. This doesn't support the requirement that users may desire to control certain parameters of the query so to customize the results to their current needs. Such needs may also be changing over time. For example, a Massachusetts user may at a time be interested in buy-
ing electronic goods. Then he would set his preference to be $\text{categoryID} = \text{“electronic”}$ and $\text{buyer\_state} = \text{“MA”}$ so to obtain corresponding recommendations based on Query 2 each time he logs into the auction system. Later when he plans to buy other stuff, he may change his preference accordingly. There typically exist large numbers of users with different and changing preferences. A static continuous query clearly cannot handle this requirement.

On the other hand, to treat each user request as a separate query is also inefficient. First, it creates more burden in query specification. For example, the network administrator may want to monitor the 20-minute traffic (Query 1) between a particular srcIP and every destIPs for half an hour because a problem on this srcIP has been reported. Using the separate-query model, the same query will be repeatedly issued for every time unit within the half hour. This would also require the system to optimize similar queries individually, thereby wasting energy.

To handle these challenging requirements in streaming groupby processing, we propose parameterized streaming groupby query (or PSGB query). A PSGB query has a basic structure to be specified at query registration time, and dynamic parameters to be specified at runtime by user requests. Hence, a PSGB query represents potentially infinite number of regular (i.e., non-parameterized) groupby queries that are unknown at compile time. In this paper, we consider the processing of PSGB queries that may contain the following dynamic parameters: aggregate functions, selection predicates on grouping attributes, windows, and request repeat patterns. The PSGB query produces results only upon user requests (i.e., pull execution model [3, 8]).

**PSGB Query that incorporates Query 1 in Figure 1**

```sql
SELECT srcIP, destIP, <agg-func-list>
FROM Packets
WHERE collectorID = ‘B’ and <predicates-on-grouping-attr>
GROUP BY srcIP, destIP
WINDOW <window-length>
REPEAT <repeat-counts>, <repeat-interval>
```

**User Request:** [SUM(len); srcIP=216.239.37.4; 20 Mins; (100, 1 Min)]

Figure 2: Example PSGB Query and User Request.

Figure 2 shows an example PSGB query based on Query 1 in Figure 1 and a user request that specifies the aggregate function SUM(len), the selection predicate srcIP=216.239.37.4, and the 20 minutes window. It also requires the query to continuously produce the result based on this specification every minute for 100 minutes.

1.2 Issues with PSGB Query Processing

The execution of the regular groupby operator involves two main operations: 1) grouping that classifies tuples into groups based on their grouping attribute values, and 2) aggregation that computes the result for each group using the aggregate function. Existing work has been focused on support these two operations. In static databases, a sort-based or a hash-based approach is used [13] to expedite the grouping operation. To optimize the aggregation, frequently used aggregate results may be materialized to save re-computation costs [14, 19]. Prior work in stream contexts [23] has studied how to share the aggregate results among statically specified streaming groupby queries differing only in their grouping attributes.

The PSGB operator differs from the regular groupby operator in that it needs to additionally conduct the selection operation that selects the data for producing results based on dynamic selection predicates. In our targeted applications, large and quickly-evolving groupby states (caused by rapid data arrivals), and high volumes of user requests with selective selection predicates concerning varying sets of grouping attributes can be observed. Hence, to efficiently perform the selection operation is essential to achieving good PSGB query performance. The existing techniques become largely inapplicable in this new problem domain due to these outstanding challenges, as discussed below.

First, the sort-based groupby tends to be inappropriate due to the potentially high sort overhead for streaming data. Second, traditional hash index, be it on a single attribute or on multiple attributes, may not be flexible enough to support widely-varying dynamic selection predicates (see discussion in Section 3.1). Therefore, a lightweight yet flexible indexing method is needed. Moreover, due to the quickly-evolving groupby state, and varying selection predicates and aggregate functions potentially involved in user requests, most aggregate results are usually not used frequently enough to justify the materialization. Even if they are chosen to be materialized, the index is still needed to expedite the search over materialized results based on selection predicates.

In summary, an index mechanism that can effectively organize the quickly evolving groupby state to support efficient lookup of the data based on various selection predicates is the most critical technology to be developed for PSGB query processing. Moreover, to withstand the fluctuations in data and query workloads, the groupby operator should be able to quickly observe the changes at runtime and then tune the index accordingly. In streaming data processing, it is impractical to stop the query execution to adjust the operator state. Therefore, the index solution must be amendable to enable efficient on-line migration [24] from one index structure to another. To the best of our knowledge, no existing work has focused on index design and tuning for streaming groupby operators. In this paper, we describe the first solution to this problem.

1.3 Our Approach: Groupby Index Tuning

Our index tuning approach involves the interleaved execution of three modules - index selection, index assessment and index migration. Before query execution starts, the index selection module either derives an optimal (or near-optimal) index configuration based on the estimated workload or simply applies no index to the groupby state. Once the execution starts, the index assessment module will be periodically executed. During each of its each run, it first invokes the index selection module to derive the optimal (or near-optimal) index configuration based on the up-to-date workload statistics collected at runtime. Then the cost of using the new configuration is compared with the cost of using the current one based on the new workload. If the new cost plus the potential migration cost is lower than the current cost, the index migration module is invoked to migrate the groupby state to the new index structure.

In this paper, we focus on the solution to index design and index selection because this is the first and most critical step
in index tuning. As observed by our experiments (Section 7.3), major overhead will be incurred if an inefficient index selection algorithm is used, making it infeasible in practice. Our contributions of this work are summarized as follows.

1. We propose parameterized streaming groupby query (or PSGB query) to capture the needs for providing customized results based on widely-varying user requests. Based on a comprehensive analysis of the PSGB processing problem, we identify the immediate need for an effective index mechanism that is tunable for evolving query/data workloads.

2. We propose the memory-efficient and easy-to-maintain IMP index solution for PSGB operator. The IMP index can be flexibly configured to benefit various requests (Section 3).

3. We design the EPrune index selection algorithm with a pruning technique (Section 4). EPrune is guaranteed to find the optimal IMP configuration. And by properly pruning candidates, its complexity can be significant reduced, sometimes more than ten-fold.

4. To suit the efficiency needs that are more important for online index tuning than guaranteed optimality, we design a time-efficient greedy index selection algorithm named RGreedy and equip it with three alternative search heuristics (Section 5). RGreedy is shown to find the near-optimal IMP configuration with observed polynomial complexity even in large search spaces.

5. Our experiment results conducted in a real stream processing system (Section 7) show that the IMP index always wins over the state-of-the-art hash index methods. RGreedy with the Hybrid heuristic finds the optimal IMP configurations in a variety of test cases with xx% confidence. For large search spaces, when EPrune takes hours to finish, RGreedy always terminates within seconds. Moreover, the groupby operator with timely and appropriate index tuning outperforms the operator with a fixed index configuration.

6. Our index selection algorithms are shown to be generally applicable in scenarios where aggregate results are chosen to be materialized (Section 6). The proposed techniques can also be used for partial-match retrieval [2] (i.e., pure selection queries) over streaming data.

2. PRELIMINARIES

2.1 Parameterized Streaming Groupby Query

In this paper, we consider the processing of PSGB queries as shown in Figure 3. The dynamic parameters are underlined and all the other query constructs are statically specified at compile time. The WINDOW clause specifies a suffix window [3] that ends at the current time. It indicates the length of the data history to be queried. The predicates in the WHERE clause are all selection predicates. The static predicates involve only non-grouping attributes and the dynamic predicates are conjunctive equality selections on the grouping attributes. The REPEAT clause specifies how many times (repeat-counts) how often (repeat-interval) the user requests should be repeated. If a user request doesn’t specify the window length, the default, largest allowed window is assumed. Similarly, the default request repeat pattern is (1, 0), i.e., to execute the request only once.

Consider the example SPGB Query in Figure ???. The where the grouping attributes. The REPEAT clause specifies how equality the dynamic predicates are conjunctive speciﬁes in the WHERE clause are all selection predicates. The attributes involved in the DSP speciﬁed in the corresponding requests are denoted by their respective names. They are also called speciﬁed attributes. Any grouping attribute not involved in DSP is represented by a * (wildcard).

Consider the PSGB Query 1 in Figure 2. The DSP “srcIP = 216.239.37.4" and the DSP “srcIP = 207.46.250.19” both match the selection pattern <srcIP, *>. Given N grouping attributes, they are in total $2^N$ distinct selection patterns.

2.3 Execution of PSGB Operator

We assume the following execution logic for the PSGB operator. It maintains a state to hold all tuples residing in the current suffix window. As a new tuple $t$ is received, it is first inserted into the state. Then, if the window is time-based, the timestamp of $t$ is used to purge the tuples in the state that have expired from the window (we assume the input tuples are received in the order of their timestamp). If the window is count-based, the oldest tuple in the state will be removed. When a pull request is received, the groupby state is probed and the aggregate results are produced based on matching tuples. If the aggregate results had been materialized, then those would be retrieved directly.

select <group-attr-list>, <agg-func-list>
from <stream-name>
where <static-preds> and <dynamic-preds>
group by <group-attr-list>
window <window-length>
repeat <repeat-lengths>, <repeat-intervals>

Figure 3: PSGB Query Specification.
3. INDEX DESIGN AND SELECTION

3.1 Requirements on Index Design

Since we focus on equality selection in user requests, a hash-based index appears to be a good fit for organizing the groupby state [13]. However, traditional one-level hash index is only appropriate when the hash keys form a subset of the search keys, i.e., the specified attributes. Otherwise a full scan of the groupby state is required. Hence it is not flexible enough to handle selections that involve varied or even disjoint sets of grouping attributes. On the other hand, to build multiple indexes is also undesirable because each tuple would then require multiple references, causing potentially high memory overhead. And memory is a highly precious resource since the groupby state should be maximally kept in memory to real-time query response [5]. The maintenance cost of multiple indexes regarding streaming data may also be considerable. Therefore, we need a new index solution to improve the performance of the PSGB operator by balancing the request processing with the index maintenance cost. The following design guidelines apply:

1. The index should benefit as many requests as possible.
2. The index structure should require minimal maintenance effort when processing data updates.
3. The index structure should be memory-efficient to be maintained in main memory.
4. The index should be lightweight to be easily migratable when the workload experiences significant changes.

3.2 IMP: Importance-Based Partition Index

In view of these requirements, we employ a multi-level prioritized hash index named IMP index (for Importance-based Partition index), as shown in Figure 4. A similar structure has been used by Aho and Ullman [2] for processing partial-match queries in information retrieval systems.

![IMP Index Diagram](image)

**Figure 4: IMP Index.**

The IMP index distributes tuples in the groupby state into \(2^B\) partitions and uses a \(B\)-bit string to represent the address of each partition. \(B\) is derived according to memory constraints on the index structure. Each grouping attribute \(A_i (1 \leq A_i \leq N)\) is assigned \(b_i\) contiguous bits such that \(0 \leq b_i \leq B\) and \(\sum b_i = B\) with \(1 \leq b_i \leq B\). Then \(A_i\) corresponds to \(2^{b_i}\) partitions. Henceforth, we use vector \(\langle b_1, b_2, ..., b_N \rangle\) to represent the bits assigned to the \(N\) grouping attributes \(A_1, ..., A_N\). We name it the IMP configuration. The attributes corresponding to non-zero bits are called indexed attributes.

A hash function is used to map the values of each indexed attribute into a bit string of the desired length. For an input tuple, its values of all the indexed attributes are used to compute the address of the single partition it should be placed into. Within each partition, tuples are ordered chronologically to facilitate the invalidation of tuples based on sliding window semantics. For an input request, if it doesn’t contain any wildcard, i.e., all grouping attributes are specified, a single partition needs to be probed to obtain all tuples that match the query. If it contains at least one wildcard, multiple partitions need to be probed. Figure 4 shows an example IMP configuration and the partition address computation for a tuple and a request. This request doesn’t specify attribute B. Since attribute B occupies 2 bits, \(2^2 = 4\) partitions need to be probed to answer the request.

3.3 Advantages of IMP Index

First, the IMP index is lightweight. It only stores the addresses to all partitions in a hash table. The partition addresses are computable with a very simple formula. Hence it is memory-efficient and easy-to-maintenance, especially compared to the tree-structured indexes. It is also easily-migrated, with no need to rebuild any auxiliary structures.

Second, the IMP index simplifies the index selection process since it unifies the two-step decisions on which attributes to index and how to index them into a single decision on how many address bits to give to each attribute. If an attribute is assigned 0 bits, this attribute is not indexed.

The most important feature of the IMP index is that the address bit allocation among the indexed attributes affects query performance non-trivially. The more bits are assigned to an attribute, the number of partitions to be probed to answer the requests that specify conditions on this attribute will be potentially decreased. Therefore, we should assign more bits to frequently queried attributes (i.e., important attributes) to reduce the overall processing cost.

We now use an example to illustrate how the IMP index configuration affects the query performance. Assume a groupby query with two grouping attributes A and B. Suppose the requests \(\langle A, * \rangle\) and \(\langle *, B \rangle\) account for 60% and 40% of all requests respectively. Suppose we use 10 bits for partition address (\(2^{10} = 1024\) partitions in total). Let’s compare two configurations. In the first configuration, we assign 6 bits to attribute A and 4 bits to B. Then we need to examine 16 partitions to answer request \(\langle A, * \rangle\) and 64 partitions to answer request \(\langle *, B \rangle\). On average 0.6 \(16 + 0.4 \cdot 64\) = 36 partitions need to be examined to answer a request. In the second configuration, we assign 4 bits to attribute A and 6 bits to B. On average 0.6 \(64 + 0.4 \cdot 16\) = 45 partitions need to be checked to answer a request. We can see that since attribute A is queried more frequently than B, assigning more bits to A achieves less overall probe cost.

3.4 Index Selection Problem Definition

The appropriate configuration of the IMP index is the key to achieving good groupby processing performance. The index selection task is then to select the IMP configuration that achieves minimum or close-to-minimum query processing cost. We first define two terms that will be used in our
index selection problem definition.

**Definition 3.1. Request Occurrence Frequency.** The occurrence frequency of a request pattern \( r_i \) in a workload \( W \) equals \( \frac{1}{W} \), with \( L \) being the number of requests in \( W \) that match \( r_i \), and \( R \) being total number of requests in \( W \).

In addition, we use unit processing cost (UPC) as the measurement of IMP configurations. UPC is defined to be the cost for processing the data tuples and the pull requests received within a time unit. Clearly the IMP configuration with the minimum UPC for the given workload is the optimal index configuration. Henceforth, we will use the terms processing cost and unit processing cost interchangeably.

Our index selection problems is defined as follows. Given a query with \( N \) grouping attributes \( A_i \) \( (1 \leq i \leq N) \) and \( B \) address bits \( (i.e., \ 2^B \) partitions) assumed by memory constraints, our goal is to select an IMP configuration that minimizes the cost for processing the given groupby workload with the following parameters (all are average values):

- data arrival rate \( \lambda_d \);
- request arrival rate \( \lambda_r \);
- request occurrence frequencies (Definition 3.1).

In the following, we describe two index selection algorithms that are suitable for different index selection and tuning scenarios.

4. **EPRUNE: PRUNED EXHAUSTIVE SEARCH**

We first describe the EPrune algorithm that is guaranteed to find the optimal IMP configuration by conducting an optimized exhaustive search.

4.1 **Algorithm Description**

The algorithm is composed of two tasks: (1) candidate construction that constructs IMP configurations, and (2) cost evaluation that evaluates the given candidates and returns the one with the minimum processing cost. To be memory-efficient, the algorithm iteratively generates each IMP configuration and directly feeds it to the cost evaluation module, i.e., in a pipelined fashion. Hence these two tasks are interleaved.

**Candidate construction.** We construct all possible IMP configurations by progressively including more attributes. Figure 5(a) depicts the process. The \( y \) axis represents the attributes included and the \( x \) axis represents the number of bits assigned to the respective attribute sets. Entry \( (i,j) \) contains all possible IMP configurations \( <b_1, b_2 \ldots, b_N> \) satisfying the following conditions: 1) \( 0 \leq b_k \leq B \) for \( 1 \leq k \leq j \); 2) \( b_m=0 \) for \( j < m \leq N \); and 3) \( \sum_{j=0}^{N} b_i = i \).

Therefore, all IMP configurations over attribute set \( <A_1, A_2, \ldots, A_N> \) using \( B \) bits can be found at entries \((B,k)\) with \( 1 \leq k \leq N \). Candidates in these entries achieve the finest partition by using all \( B \) bits. This is a prerequisite to achieving minimum processing cost under the \( B \)-bit constraint. Hence, only these candidates will be input into the cost evaluation module.

The arrow in the figure indicates the order in which these candidates are generated. To construct candidates for \((B,j)\) \((0<j\leq N)\), all candidates of \((p, j-1)\) \((0\leq p \leq B-1)\) are iterated over and \( b_j \) in each of them is set to be \((B-p)\), as shown by the shaded blocks in Figure 5(a). (SAY IT IN WORDS: iterate over horizontal at level \( j-1 \) and set \( b \))

**Figure 5: Algorithm Search Spaces.**

**Cost Evaluation.** The cost evaluation module evaluates the processing cost for each constructed IMP configuration and keeps the optimal IMP configuration found thus far.

Equation 1 computes the groupby processing cost. It includes the tuple processing cost \( C_T \) and the request processing cost \( C_R \). \( C_T \) includes the cost for hashing the indexed attributes \( (C_{hash,T}) \), inserting tuples into the index \( (C_{insert}) \) and deleting tuples from the index \( (C_{delete}) \). \( C_R \) includes the cost for hashing the indexed attributes specified in the requests \( (C_{hash,R}) \), probing the resultant partitions \( (C_{probe}) \) and generating aggregate results \( (C_{aggregate}) \).

\[
C = C_T + C_R = C_{hash,T} + C_{insert} + C_{delete} + C_{hash,R} + C_{probe} + C_{aggregate} \tag{1}
\]

Among these costs, \( C_{insert}, C_{delete} \) and \( C_{aggregate} \) are independent of the IMP configuration. Hence the cost evaluation module only needs to compute the IMP-relevant cost (denoted as \( C_V \)) for all candidates. \( C_V \) is the sum of \( C_{hash,T}, C_{hash,R} \) and \( C_{probe} \), as computed by Equation 2 using the notations defined in Table 1.

\[
C_V = C_{hash,T} + C_{hash,R} + C_{probe} = \lambda_d N_A C_h + \lambda_r \sum_{r_i \in R} (N_{A,r_i} C_h + \frac{\lambda_d W}{2^{B_{r_i}}} C_c) \tag{2}
\]

For an incoming tuple \( t \), \( N_A \) hash operations need to be conducted to find the exact partition to insert \( t \). Hence \( C_{hash,T} \) equals \( \lambda_d N_A C_h \). For a pull request \( r_i \), its associated hash cost depends on the number of indexed attributes specified in \( r_i \). As \( r_i \) ranges over \( R \), the set of all pull requests that arrive within a time unit, \( C_{hash,R} \) equals \( \lambda_R \sum_{r_i \in R} N_{A,r_i} C_h \). The probing cost associated with \( r_i \) equals

**Table 1: Notations.**
the total number of partitions probed, \(2^B - B r_i\), times the average number of tuples in each partition, \(\frac{\lambda W}{2}\), and then times the value comparison cost \(C_c\). As \(r_i\) ranges over \(R\), \(\text{Config}_{\text{probe}}\) equals \(\frac{\lambda W}{2} \sum_{r_i \in R} r_i C_c\).

4.2 Pruning Strategy

If an attribute is queried infrequently enough so that the overhead for indexing it exceeds the gained probe cost reduction, this attribute will not be included into the index. If those attributes can be detected and pruned before the search starts, the search space may be significantly shrunk. We achieve this goal by employing a benefit function. Definition 3 defines the attribute occurrence weight to be used in the benefit function.

**Definition 4.1. Attribute Occurrence Weight.** The occurrence weight of an attribute \(A\), denoted as \(\text{OW}(A)\), is defined to be the sum of the occurrence frequencies of all requests in which a non-wildcard value is specified for \(A\).

By indexing an attribute \(A\), the minimum probe cost reduction is \(\text{OW}(A) \cdot \lambda W \sum_{A_h} C_c\) (assuming \(A\) is assigned one bit). The increased hash cost is \(\text{OW}(A) \cdot \lambda_h C_h\). Hence the benefit of \(A\) is as computed by Equation 3. The attributes with negative benefit values can then be removed from consideration.

\[
\text{Benefit}(A) = \text{OW}(A) \left( \lambda \sum_{A_h} C_c - \lambda_h C_h \right) - \lambda_h C_h \tag{3}
\]

This is a conservative estimation since it uses the minimum probe cost reduction (i.e., the gains). Therefore, the pruned attributes are assured not to appear in the optimal IMP configuration. Hence the optimality of EPrune is still guaranteed. The pseudo code of EPrune is shown in Algorithm 1.

**Algorithm 1 EPrune Algorithm**

**Input:** Attribute set \(S_A\), integer \(B\), request pattern set \(S_R\)

**Output:** An IMP configuration with minimum cost

\[
\text{optimal}_\text{config} := \langle 0, ..., 0 \rangle; \quad * * \text{ no attr. is indexed. } */
\]

\[
\text{optimal}_\text{cost} := \text{Cost} (\text{optimal}_\text{config}, S_R);
\]

\[
S_A, \text{RemoveNonBeneficialAttr}();
\]

Initialize candidate construction module \(cc\) by \(S_A\) and \(B\);

while \(cc.\text{HasMoreConfigs}()\) do

\[
\text{new}_\text{config} := cc.\text{GetNextConfig}();
\]

\[
\text{new}_\text{cost} := \text{Cost} (\text{new}_\text{config}, S_R);
\]

if \(\text{new}_\text{cost} < \text{optimal}_\text{cost}\) then

\[
\text{optimal}_\text{config} := \text{new}_\text{config};
\]

\[
\text{optimal}_\text{cost} := \text{new}_\text{cost};
\]

end if

Return \(\text{optimal}_\text{config}\);

4.3 Complexity Analysis

In this paper, we define the complexity of an index selection algorithm to be the total number of IMP candidates evaluated by the algorithm.

Given \(N\) attributes and \(B\) address bits, any IMP configuration that has \(\leq \text{Min}(N, B)\) attributes to share \(B\) bits is a potential index solution. To index \(k\) (\(1 \leq k \leq \text{Min}(N, B)\)) attributes, the total number of distinct IMP configurations equals the total number of ways to place \((k-1)\) marks among \((B-1)\) positions to divide a \(B\)-bit string into \(N\) pieces, which is \(\binom{B - 1}{k - 1}\). There are \(\binom{N}{k}\) ways to select \(k\) attributes out of a set of \(N\) attributes. Hence the complexity of EPrune can be computed by Equation 4.

\[
\text{Complexity} (\text{EPrune}) = \sum_{k=1}^{\text{Min}(N, B)} \binom{N}{k} \binom{B - 1}{k - 1} \tag{4}
\]

By employing the pruning strategy, with \(M\) attributes being pruned, the complexity of EPrune becomes

\[
\sum_{k=1}^{\text{Min}(N - M, B)} \binom{N - M}{k} \binom{B - 1}{k - 1}.
\]

This can lead to huge search savings. For example, when \(B=16\) and \(N=10\), the complexity of EPrune is 2,042,975. Suppose \(M=2\), the complexity is reduced to 245,157, by a factor of 10.

5. RGREEDY: GREEDY ALGORITHM

We can see that for a fixed \(B\) value (e.g., assume \(B=16\)), when \(N_r\) = \(N - M\) is small, the optimal IMP configuration could be quickly found by EPrune. For example, the complexity when \(N_r=4\) is 969. However, as the value of \(N_r\) increases, the complexity increases precipitously. As shown before, when \(N_r=10\), the complexity of EPrune becomes 2,042,975.

Then EPrune may take hours or even days to finish. In streaming environments, data/request statistics may often experience unpredictable changes. Therefore, it is not practical to spend a major time to search for an optimal index configuration that may become sub-optimal shortly after. In response, time-efficient algorithms are needed to quickly find a near-optimal configuration even in large search spaces. Below we introduce a heuristic-based greedy algorithm named RGreedy.

RGreedy doesn’t guarantee the optimality but are shown to be useful for a huge variety of practical cases. And it is acceptable to trade the optimality for timeliness in many stream processing applications for which real-time answers are critical.

5.1 Algorithm Overview

The basic idea of the RGreedy algorithm is to first rank each grouping attribute by the benefit that may be obtained from indexing that attribute (i.e., the *importance* of the attribute). Then the algorithm progressively includes the attribute with the next highest ranking into consideration for inclusion into the index. At each step, the algorithm constructs the new IMP candidates, assuming all attributes considered at this step are being indexed. If the optimal IMP configuration among these new candidates achieves less cost than the optimal IMP configuration derived from all previous steps, the algorithm will continue. Otherwise, the algorithm terminates and the IMP configuration found to be best thus far is returned. The intuition is that if it is not beneficial to index an attribute, to index a less important attribute is unlikely to achieve cost savings. The pseudo code of RGreedy algorithm is shown in Algorithm 2.

**Complexity.** The RGreedy algorithm progressively considers attributes. It only searches through the IMP configurations in which all considered attributes are indexed. Hence RGreedy has a significantly lower complexity than EPrune, as shown in Figure 5(b). Equation 5 shows the worst case complexity of RGreedy. Since RGreedy stops whenever no further cost reduction can be achieved by considering one
Algorithm 2 RGreedy Algorithm

Input: Attribute set $S_A$, integer $B$, request pattern set $S_R$, heuristic $H$

Output: An IMP configuration with enumerated minimum cost

```
optimal_config := <0, ... , 0>;
optimal_cost := Cost(optimal_config, $S_R$);
$L_A := Rank(H, S_A);$ 
$S_I := \emptyset$;
while $L_A\text{.Has MoreAttr()}$ do
    $S_I := S_I + L_A.$GetNextAttr();
    $S_I := GenerateCandidatesGreedy(S_I, B)$;
    while $S_C.$HasMoreConfig() do
        new_config := $S_C.$GetNextConfig();
        new_cost := Cost(new_config, $S_R$);
        if new_cost < optimal_cost then
            optimal_config := new_config;
            optimal_cost := new_cost;
        end if
    end while
end while
Return optimal_config;
```

more candidate, and the optimization technique proposed for EPrune is also applicable to RGreedy; we find that the complexity of RGreedy is usually much lower than the worst case complexity (see Section 7.3).

$$\text{WorstCaseComplexity}(\text{RGreedy}) = \sum_{i=1}^{\text{Min}(N, B)} \left( \frac{B - 1}{i - 1} \right)$$

Measuring attribute importance. The effectiveness of the RGreedy algorithm clearly relies on the order in which the attributes are being considered. Such order is determined by a function for ranking the attribute importance. This indexing also determines how fast the algorithm terminates. In addition, as more attributes are being included into the index, the hash cost increases. An ill-designed importance measure may cause the algorithm to stop before most important attributes are even considered. Therefore, the order also affects the quality of the configuration found by the algorithm. We have designed several ranking heuristics to estimate the attribute importance. Below we will introduce two single-criterion heuristics and a hybrid heuristic.

5.1 Occurrence Weight Leading Heuristic

Our first heuristic ranks the importance of each attribute by how frequently it occurs in requests, i.e., the occurrence weight of the attribute (Definition 4.1). Hence it is named the occurrence weight leading (OWL) heuristic. The intuition is that to index the attributes that occur in a large number of requests is likely more beneficial.

Since the probe cost is usually the dominating cost, in all examples we show henceforth, we only compare the probe costs of candidate IMP configurations. And we use the average number of partitions to be probed for processing a single request as the indicator of the probe cost of each IMP configuration.

Example 1. Consider the request statistics in Table 2. The occurrence weights of attributes A, B and C are 0.5, 0.4 and 0.2 respectively. As shown in Table 3, attribute A is considered in the first iteration since it has the highest occurrence weight. A single IMP configuration is available. In the second iteration, attribute B is included. Considering both A and B, three configurations are produced and <3,1,0> has the lowest probe cost. In the third iteration, attribute C is included. Again, three candidates arise and <2,1,1> is the best. Now that all the three grouping attributes have been considered, the algorithm stops and returns configuration <2,1,1>. It corresponds to the optimal configuration found by the EPrune algorithm. While EPrune needs to examine 15 candidates, the RGreedy with OWL finds the optimal configuration by only checking 7 candidates.

Table 2: Example 1 – Request Statistics.

<table>
<thead>
<tr>
<th>Request Pattern</th>
<th>Occurrence Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;A, B, *&gt;</td>
<td>0.4</td>
</tr>
<tr>
<td>&lt;*, B, *&gt;</td>
<td>0.3</td>
</tr>
<tr>
<td>&lt;*, *, C&gt;</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 3: Example 1 – OWL Execution.

<table>
<thead>
<tr>
<th>Step</th>
<th>Next Attr</th>
<th>Configuration</th>
<th>Probe Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>&lt;3,0,0&gt;</td>
<td>8.5</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>&lt;1,2,0&gt;</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;3,1,0&gt;</td>
<td>6.5</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>&lt;3,1,2&gt;</td>
<td>6.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;3,2,1&gt;</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;3,2,1&gt;</td>
<td>5.8</td>
</tr>
</tbody>
</table>

Table 4: Example 2 – Request Statistics.

<table>
<thead>
<tr>
<th>Request Pattern</th>
<th>Occurrence Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;A, *, *&gt;</td>
<td>0.3</td>
</tr>
<tr>
<td>&lt;A, B, *&gt;</td>
<td>0.4</td>
</tr>
<tr>
<td>&lt;*, *, C&gt;</td>
<td>0.3</td>
</tr>
</tbody>
</table>

5.1.2 Request Coverage Leading Heuristic

In view of the above shortcoming of the OWL heuristic, we propose another heuristic, namely request coverage leading
(RCL), that instead considers the request coverage of the attributes. The intuition is that the index is likely to be beneficial by indexing the attributes that together cover a majority of requests.

**Definition 5.1. Covered Request and Remaining Request Coverage.** A request \( r \) is said to be covered by an index if at least one attribute specified in \( r \) is included in the index. The remaining request coverage of an attribute \( A \), denoted as \( \text{RRC}(A) \), is defined to be the sum of the occurrence frequencies of all requests that are not covered by the index and specify a non-wildcard value on \( A \).

While the attribute occurrence weights are static, the remaining request coverage of an attribute \( A \) must be recomputed as additional attributes are being included into the index.

Table 5 shows the execution of the RGreedy algorithm using RCL given the statistics in Example 2. We see that while OWL missed the optimal configuration, RCL finds it.

<table>
<thead>
<tr>
<th>Step</th>
<th>RRC</th>
<th>Next Attr.</th>
<th>Configuration</th>
<th>Probe Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A: 0.7, B: 0.4, C: 0.3</td>
<td>&lt;4.0,0&gt;</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C: 0.3, B: 0</td>
<td>C</td>
<td>&lt;3.0,1&gt;</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Table 5: Example 2 – RCL Execution.

The RCL heuristic works well for the cases when frequently queried attributes are completely correlated with each other. Hence it overcomes the shortcomings of OWL. However, it still does not guarantee to always find the optimal IM configuration. It fails when a frequent attribute correlates with both frequent and infrequent attributes and is shadowed by the infrequent attributes in terms of request coverage. Example 3 below illustrates this scenario.

**Example 3.** Consider the statistics in Table 6. Attribute \( B \) is a frequent attribute. It co-occurs with both \( A \) (frequent) and \( C \) (infrequent). During the search, after \( A \) is included, \( \text{RRC}(B) \) is reduced from 0.65 to 0.25 so it becomes less than \( \text{RRC}(C) \), which is 0.3. Therefore, \( C \) will be considered next with best configuration \( <3,0,1> \). After \( C \) is included, \( \text{RRC}(B) \) is reduced to zero. Then \( B \) is never considered by the algorithm. However, the optimal configuration is \( <2,2,0> \) since including \( B \) instead of \( C \) actually benefits more queries. Interestingly, OWL is able to find this optimal configuration that RCL missed.

<table>
<thead>
<tr>
<th>Request Pattern</th>
<th>Occurrence Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;A, *, *&gt;)</td>
<td>0.3</td>
</tr>
<tr>
<td>(&lt;A, B, *&gt;)</td>
<td>0.4</td>
</tr>
<tr>
<td>(&lt;*, B, C&gt;)</td>
<td>0.25</td>
</tr>
<tr>
<td>(&lt;*, *, C&gt;)</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 6: Example 3 – Request Statistics

### 5.1.3 Hybrid Heuristic

It can be seen that the OWL and the RCL heuristics complement each other. We hence propose a heuristic that combines them, named Hybrid heuristic. The idea is to run the greedy search once using the OWL heuristic and then using the RCL heuristic. After that we compare the final configurations suggested by these two runs. The one with the smaller cost will be selected as final decision. By applying the Hybrid heuristic, we are able to find the optimal configurations for Examples 2 and 3, while either OWL or RCL will miss one of them respectively.

**Optimization by reusing cost computations.** If we blindly run the algorithm with OWL and the RCL heuristics and then compare the results produced by both of them, we may repeat many intermediate computations. First, the two heuristics always consider the same attribute in the first iteration because in this iteration, the OW value of each attribute equals its RRC value. In addition, in later iterations, the two heuristics may consider the same sets of attributes. Example 1 is an extreme example here since both OWL and RCL would consider attributes \( A, B \) and \( C \) in turn. Hence running one of them would be enough.

To reuse computations in using Hybrid heuristic, first, we check the attribute consideration orders of OWL and RCL. If the two orders are the same, we only run RGreedy with OWL. Otherwise, we first run RGreedy with OWL and keep the optimal IMP configuration for each iteration with its cost into a hash table. The hash key is the ID of the attribute set. Then we run RGreedy with RCL. During this run, in each iteration, we first check the hash table to see whether the given attribute set has been considered in the first run. If yes, we skip the evaluation and directly use the result.

**Analysis.** Without reusing computations, the complexity of using the Hybrid heuristic equals the sum of OWL complexity and RCL complexity. By reusing computations, it becomes the worst case complexity only when the two heuristics consider totally different sets of attributes. In the best case when OWL and RCL consider the attributes in the same order, Complexity(Hybrid) = Complexity(OWL) + Complexity(RCL).

### 6. Generalizations

#### 6.1 Dealing With Narrow Attribute Domains

The EPrune and RGreedy algorithms implicitly assumed that the number of distinct values of any indexed attribute \( A_k \) is at least equal to \( 2^w \) with \( b_k \) being the number of bits assigned to \( A_k \). However, this is not always the case. For example, the gender attribute only corresponds to two values, male and female. Hence at most one bit should be assigned to it. If the gender attribute is queried very often, it may be assigned more than one bit by the algorithms described so far. This may lead to non-optimal index configurations. Below is such an example.

**Example 5.** Consider the request statistics in Table 2. The optimal configuration found when assuming no partition limits is \( \{2, 1, 1\} \). Suppose domain of attribute \( A \) only corresponds to two values. Then one address bit will never be used. Tuples in the state only occupy 8 instead of 16 partitions. The average number of tuples being probed for processing a single request becomes \( c_3 = 3.8 \times \frac{1}{8} = 0.475A_dW \).

If the algorithm takes the narrow attribute domain into consideration, the optimal configuration then becomes \( \{1, 2, 1\} \) with cost \( c_2 = 6.2 \times \frac{W}{8} = 0.388A_dW \). Given that the groupby state usually contains large numbers of tuples (i.e., a large \( A_dW \) value), \( c_3 \) should be lower than \( c_2 \) most of the time.

\( \lambda \)
To solve this problem, we set the upper limit on the number of bits assigned to each attribute $A_i$ ($1 \leq i \leq N$) to be $b_{max}^i = \lceil \log_2 D_{V_i} \rceil$ with $D_{V_i}$ denoting the number of distinct values of $A_i$. We revise EPrune and RGreedy algorithms to construct the configuration candidates subject to these limits. This way the optimality of the EPrune algorithm is still guaranteed.

**Theorem 6.1.** Given $N$ grouping attributes $A_1$, ..., $A_N$, $B$ address bits and statistics on $D_{V_i}$ for each $A_i$ ($1 \leq i \leq N$), the IMP configuration output by the revised EPrune algorithm has the lowest UPC among all possible IMP configurations.

### 6.2 Dynamic Window Specification and Dynamic Aggregate Functions

So far we have assumed that the WINDOW clause in the groupby query is fixed (see query specification in Figure ??), i.e., all pull requests are evaluated over a fixed-length sliding window. However, dynamic window specification is sometimes required so to allow users to specify the history they would like to query on. Then different pull requests may specify different window sizes. For example, the recommendation service of the online transaction system (Section 1.1) will be more appealing if it can provide recommendations according to a user-defined window. We now show that our approaches can be naturally extended to handle such dynamic window specification.

First, the processing of a request is changed so that instead of probing all tuples in the corresponding hash partitions, now the probing will start from the tail (i.e., the latest-received tuple) of each partition, and stop once the first expired tuple is encountered. Then if the window specified in the pull request is significantly smaller than the default window assumed in the query, only a small portion of the data in the groupby state needs to be examined. Second, the index selection cost model and algorithms need to be adjusted accordingly, as illustrated by Example 6.

**Example 6.** Consider the request statistics in Table 4. Under the fixed-window assumption, the optimal configuration is $<3, 0, 1>$. Let’s now consider the varying-window case such that the average window sizes of request patterns $<A, \ast, \ast>$ and $<A, B, \ast>$ are both $\frac{1}{3}$ of the default window, while for request pattern $<\ast, \ast, C>$, it equals the default window. We assume that tuples in each hash IMP partition are uniformly distributed w.r.t. their timestamps. Then the optimal configuration under the varying-window case for this example is $<1, 0, 3>$, with average # of partitions per request being $(0.3 + 0.4) \times 8 \times \frac{1}{3} + 0.3 \times 3 = 1.16$. This is because although attribute $A$ is queried significantly more often than attribute $C$, each time only a small tail portion of the bucket needs to be probed. Hence, even if we distribute $A$ values into a smaller number of partitions, the probing overhead doesn’t grow considerably. By giving attribute $C$ more address bits, the probing cost per request drops significantly.

Therefore, given statistics on both occurrence frequencies and average window sizes of request patterns, we modify the cost model to use revised request pattern frequencies. The revised request pattern frequency is defined to be the original request pattern frequency times a reduction factor, which is the ratio of the average window size of this request pattern to the default window size. Before running each index selection algorithm, we compute the revised request pattern frequencies as input.

In addition, the aggregate functions can also be dynamically specified. The statistics on aggregate functions will not affect our index selection decisions. It may be used to determine whether the aggregate results need to be pre-computed or not, as will be described in Section 6.3 below.

### 6.3 Supporting Pre-computed Aggregates

The algorithms described so far assume that no aggregate results are pre-computed, which is a common setting as discussed in Section 1.2. On the other hand, it may be worthwhile to pre-compute the aggregate results if the request arrival rate is significantly higher than data arrival rate and most requests are less selective. Similar to the aggregate processing in prior works [16, 23], decisions on whether and what to pre-compute are based on the benefit analysis. While the above is an orthogonal issue, once the pre-computation decision is made, our approach can easily be extended as below to incorporate this option.

First, the index selection cost model is changed to include the cost for maintaining the pre-computed results. Second, the probe cost is changed. Previously each entry in the hash partition represents a single tuple. Now it represents the aggregate result for each finest group (more advanced data structures may be used for each entry, such as the techniques in [3]). The search within each partition should stop once the matching entry is found. Hence the average search cost per partition is half number of the entries in the partition.

Using this revised cost model, the proposed index selection algorithms are still applicable with no modification.

### 7. EXPERIMENTAL STUDY

#### 7.1 Experimental Setup

We have implemented the pull-based groupby operator with index tuning component in our Java-based continuous query system. We have conducted an extensive experimental study to explore the effectiveness of the index selection algorithms and the index tuning approach. The test machine has a 3GHz Intel(R) Pentium-IV processor and a 1GB RAM, running Windows XP and Java 1.5.0.06 SDK. In all the experiments shown below, 16 address bits (i.e., $2^{16}$ hash partitions) are assumed for the IMP index.

#### 7.2 Comparing Index Methods

First, we aim to answer the question whether the IMP index is a valid solution for our problem. We run the groupby operator over a large number of pull-based groupby queries with various workloads and use three different index approaches – IMP index, single-attribute hash index on the attribute with the highest occurrence weight (denoted as $1$-Attr-Hash), and multi-attribute hash index on the attributes occurring in the most frequent request pattern (denoted as $M$-Attr-Hash).

In these experiments, the operator using the optimal IMP index always achieves the equivalent or in practically all cases, significantly outperform the other two approaches. We now show the results of one such experiment.

In this experiment, tuples and requests are generated such that each tuple is followed by a pull request (i.e., $\frac{\lambda}{\mu} = 1$). For tuples (or requests), the values of each grouping attribute conform to a uniform distribution with 2048 distinct
values. We conduct 6 runs, each over a workload containing 200,000 tuples and 200,000 requests, i.e., each tuple followed by a request. In these runs, we use 3 different request statistics that are respectively specified in Examples 1 (Table 2), 2 (Table 4) and 3 (Table 6) in Section 7.7. We also vary the window size to contain different number of tuples. Table 7 summarizes the experimental parameters. We record the total execution time of the groupby operator in each of these runs and display them in Figure 6.

We can see that, in all these runs, the groupby with the IMP index speeds up the query processing by at least 50%, in some cases even more than 90% compared to using the other two index methods. The gains increase with the window size. This is because as the groupby state becomes larger, by properly balancing the partition factors among all grouping attributes, the IMP index tends to gain more probing cost savings compared to the other two approaches.

### 7.3 Comparing Index Selection Algorithms

The effectiveness of the index tuning process largely depends on the effectiveness of the index selection algorithm being used. Hence, our second set of experiments compares different index selection algorithms we have proposed regarding how fast each algorithm terminates (i.e., efficiency) and how close each of their decisions approaches the optimal configuration (i.e., optimality).

**Random request load.** First, we run the four algorithms – EPrune, RGreedy with OWL, RCL and Hybrid heuristics respectively over randomly generated request statistics based on the normal distribution and then also on the zipf distribution. In the statistics generation, we first randomize the request patterns into an array. Then we conduct 1,000,000 pattern selections. At each selection, we use the distribution function to decide the entry in the pattern array and then select the pattern residing in that entry. Finally we summarize the occurrence frequency of each pattern within the 1,000,000 pattern selections. The reason for randomizing the request patterns at the beginning is to vary the frequent patterns in different runs for generating request statistics.

To test a large variety of workloads, for normal distribution, we vary the variance, and for zipf distribution, we vary the alpha value to control the skew factor. We also vary the tuple arrival rate and the request arrival rate in the workload and the window size in the query specification. We run the four index selection algorithms over the generated workload parameters and record the complexity (total number of IMP configurations being evaluated), the cost of the configuration being found and the search time for each algorithm. Note that RGreedy with the Hybrid heuristic evaluates each distinct IMP configuration once since it reuses computations.

For illustration purpose, we now show one experiment in which the groupby query has 8 attributes. The request statistics are generated by a normal distribution, with 8 different variances. A one-million-tuple window is assumed and \( \lambda = 10 \). Table 8 shows the execution time of the EPrune algorithm. We can see that the execution time increases with the normal variance. This is because as the normal variance increases, more request patterns have non-zero occurrence frequency. Then the cost evaluation for each IMP configuration becomes more expensive as it needs to iterate over each request pattern. In addition, within these workloads, all the grouping attributes have a non-trivial occurrence weight. Hence no attribute can be removed from consideration by the pruning strategy. Then the EPrune algorithm takes hours to finish for large normal variance value. However, the RGreedy algorithm terminates within seconds, as shown in Figure 7(a).

Figure 7(b) shows the UPC of the best IMP configuration found by each of the four algorithms over the given workloads. Here we use relative cost instead of the real cost. RCL and therefore Hybrid find the optimal configuration in all 8 cases. Hence they meet with EPrune at all points in Figure 7(b). While OWL misses all the optimal configurations, its decisions are quite close to the optimal configurations, with the no-index cost being \( 10^7 \).

Finally, from Figure 7 we can see that the Hybrid heuristic combines the advantages of OWL and RCL. And by reusing computations, its extra overhead is moderate, and in many cases even trivial. This indicates that the Hybrid heuristic should be the suggested heuristic for large search spaces.

We have also tested the four algorithms over the request statistics generated by the zipf distribution with alpha values ranging from 0.7 to 1.1 with 0.1 increment. The RGreedy algorithm with the Hybrid heuristic finds the optimal configuration for all the tested cases and terminates within seconds, while EPrune again takes hours to finish.

**Extreme request load.** Second, we show the experiment that tests the four index selection algorithms over four well-designed extreme workloads, namely, Uniform, Asc, Desc and Exclusive. In the Uniform workload, all possible request

<table>
<thead>
<tr>
<th>Run #</th>
<th>Request Load</th>
<th>Window Size (# Tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example 1</td>
<td>100,000</td>
</tr>
<tr>
<td>2</td>
<td>Example 1</td>
<td>200,000</td>
</tr>
<tr>
<td>3</td>
<td>Example 2</td>
<td>100,000</td>
</tr>
<tr>
<td>4</td>
<td>Example 2</td>
<td>200,000</td>
</tr>
<tr>
<td>5</td>
<td>Example 3</td>
<td>100,000</td>
</tr>
<tr>
<td>6</td>
<td>Example 3</td>
<td>200,000</td>
</tr>
</tbody>
</table>

Table 7: Parameters for Experiment 1.
n tuples and 200,000 requests. The migration time for the tuned groupby operator never conducts migration. That is, it always uses the IMP configuration that is optimal for the first stage. We then record the total execution time of the groupby operator, including the index assessment and migration time.

Figure 9 shows the results for two such runs, using two different window sizes respectively. In the figure, we use three different colors to mark the three execution stages.

Figure 9: Tuned Operator vs. Not Tuned Operator.

Another interesting observation (from both random and extreme experiments) not shown here is that as the window size increases, both EPrune and RGreedy take a longer time to finish, though RGreedy still terminates within minutes. This is because for the same request statistics, the larger window creates the need for indexing more attributes.

### 7.4 Performance Gains from Index Tuning

Finally, we show how the index tuning affects the query performance. In this experiment, we use a workload containing 600,000 tuples and 600,000 requests. The first, middle and the last 200,000 requests are generated respectively according to the request statistics specified in Examples 2, 1 and 3 in Section ???. Hence the groupby execution is composed of three stages, each processing 200,000 tuples and 200,000 requests.

We then run the groupby operator two times. In the first run, the operator is forced to conduct instant migration that rehashes every tuple in the state after it finishes processing every 200,000 requests. It uses the optimal IMP configuration for each of the three stages. In the second run, the operator never conducts migration. That is, it always uses the IMP configuration that is optimal for the first stage.

Figure 9 shows the results for two such runs, using two different window sizes respectively. In the figure, we use three different colors to mark the three execution stages.

Figure 9: Tuned Operator vs. Not Tuned Operator.

8. RELATED WORK

The groupby operator with index tuning achieves more than 50% reduction on the execution time compared to the one without tuning. The migration time for the tuned groupby operator is invisible in the figure because it is too short to be shown. In both sets of experiments, the time for each migration is within seconds. This is promising because it shows little migration overhead for relatively large groupby states (containing 100,000 and 200,000 tuples respectively). This indicates that if employing an efficient index selection algorithm, the index tuning can indeed help the groupby operator to achieve significant performance gains.

8. RELATED WORK

The index selection problem has been extensively studied in static databases [1, 9, 11, 15], in which data updates are much more infrequent than queries. The tools that have been built take a query workload as input and suggest a set of indexes that can maximally benefit the given workload. Index adaptation with changes in workloads means inserting a new index or deleting an existing index. We instead tackle the index selection problem in the stream context where data
updates and query requests may both arrive at high rates. Further, we target parameterized streaming groupby queries with runtime query instantiations by user requests.

Indexing in stream contexts has not yet received much attention, possibly due to the dynamic nature of the streaming data. [12] studies methods for indexing a single attribute for streaming algebra operators under the sliding window semantics and push model. The index selection driven by workloads, such as the focus of our work, is not tackled.

Our work also relates to the work on processing groupby or aggregate queries over streaming data. Existing work mainly focuses on sharing aggregate results. Their targeted query types and assumed execution models are summarized in Table 9. [23] studies aggregate result sharing among a set of streaming groupby queries that differ only in their grouping attributes. This is a direct extension of the work in static databases [16]. [3, 17, 18] focus on computation methods for streaming aggregate queries without grouped operations, i.e., one single result is produced per window. [18] proposes techniques for sharing aggregate results among consecutive sliding windows of a single aggregate query by breaking windows into time slices. [17] generalizes this idea by slicing tuples into partitions based on window and selection predicate overlaps. The computation of aggregate functions over such partitions just puts them together into a combined aggregate value. All of these works assume 1) push execution model and 2) queries to be statically specified. [3] instead investigates shared execution of pull-based aggregate queries with varying suffix windows.

<table>
<thead>
<tr>
<th>Query</th>
<th>Exec. Model</th>
<th>Dynamic Feature</th>
<th>Related Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groupby</td>
<td>push</td>
<td>no</td>
<td>[23]</td>
</tr>
<tr>
<td>Aggregate</td>
<td>push</td>
<td>no</td>
<td>[17, 18]</td>
</tr>
<tr>
<td>Aggregate</td>
<td>pull</td>
<td>dynamic window</td>
<td>[3]</td>
</tr>
</tbody>
</table>

Table 9: Related Work on Shared GB/AGG Exec.

We differ from these prior work in that we focus on processing a parameterized streaming groupby query, which itself represents a potentially infinite number of regular groupby queries to be instantiated at runtime by user requests. The groupby results for different user requests may customized using different subsets of the data in groupby state.

9. CONCLUSION

In this paper, we presented the index tuning problem for pull-based continuous groupby operators. We described the adaptive index tuning process that includes three key operations – index configuration, index evaluation and index migration. We employed a lightweight index structure, namely IMP index, that can be configured to benefit various frequent query patterns. Also, it is easy to migrate. We proposed algorithms for selecting the IMP index configuration that achieves the minimum or close-to-minimum processing cost for a given workload. We conducted an extensive experimental study in a continuous query system. Our experiment results validate the effectiveness of our index selection algorithms and index tuning approach.

10. REFERENCES


