Incremental Aggregation on Multiple Continuous Queries

Chun Jin and Jaime Carbonell

Language Technologies Institute, School of Computer Science
Carnegie Mellon University, Pittsburgh, PA 15213 USA
{cjin, jgc}@cs.cmu.edu

Abstract. Continuously monitoring large-scale aggregates over data streams is important for many stream processing applications, e.g. collaborative intelligence analysis, and presents new challenges to data management systems. The first challenge is to efficiently generate the updated aggregate values and provide the new results to users after new tuples arrive. We implemented an incremental aggregation mechanism for doing so for arbitrary algebraic aggregate functions including user-defined ones by keeping up-to-date finite data summaries. The second challenge is to construct shared query evaluation plans to support large-scale queries effectively. Since multiple query optimization is NP-complete and the queries generally arrive asynchronously, we apply an incremental sharing approach to obtain the shared plans that perform reasonably well. The system is built as a part of ARGUS, a stream processing system atop of a DBMS. The evaluation study shows that our approaches are effective and efficient on typical collaborative intelligence analysis data and queries.

1 Introduction

Aggregates are standard database operations and are widely used in data warehousing applications. They have been studied and commercialized with great success. However, recently, the emergence of the data stream processing presents new challenges to compute multiple aggregates over ever-changing data streams.

Consider a collaborative environment for intelligence analysis. An analyst wants to quickly respond to emergency situations, and discover unusual developments of special events from structured data streams. For example, an analyst concerning terrorism activities wants to detect signals of bio-terrorism attacks. So he sets up a set of continuous aggregate queries on the stream of hospital patient records to monitor possible outbreaks of contagious diseases that are possibly transmitted by bio-attack agents.

Analysts with overlapping interests and expertise share resources, results, and opinions to collaborate complex strategic and tactic tasks. In a collaborative environment, multiple analysts may set up different monitoring queries at different times. New queries are formulated when new patterns are identified as worth tracking by analysts or automatic novel intelligence tools. We expect probably hundreds or thousands of such queries to run concurrently.

Further, the wide varieties and complexity of tasks suggest the mighty richness and complexity of expected aggregate queries, including the extensive applications of complex user-defined aggregates, such as TrackClusterCenters. Tracking cluster centers is an important novel intelligence tool [10] to monitor trends and event evolutions where tuples are viewed as points in a n-dimension feature space and clustered, and the cluster centers are updated when new points arrive.

Many aggregate functions, *algebraic functions*, including MIN, MAX, COUNT, SUM, AVERAGE, STDDEV, and TrackClusterCenters, can be incrementally updated upon data changes without revisiting the entire history of grouping elements; while other aggregates, *holistic functions*, e.g. quantiles, MODE, and RANK, can not be done this way. We implemented an efficient incremental aggregation mechanism for arbitrary algebraic aggregate functions including user-defined ones. The system also supports holistic functions by reaggregating the stream history.

Consider a query A monitoring the number of visits and the average charging fees on each disease category in a hospital everyday, shown in Fig 1(a). When new tuples of patient records from the stream S arrive, the aggregates *COUNT(*)* and *AVERAGE(fee)* can be incrementally updated if *COUNT(*)* and *SUM(fee)* are stored. Our system will generate a query plan (query network), shown in Fig 2(a), to efficiently update the aggregate results.

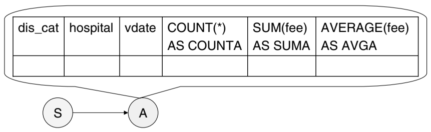
SELECT $dis\_cat, hospital, vdate,$
$COUNT(*), AVERAGE(fee)$
FROM S
GROUP BY $CAT(disease)\ AS\ dis\_cat$
$hospital,$
$DAY(visit\_date)\ AS\ vdate$

(a) Query A

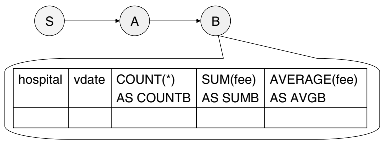
SELECT $hospital, vdate,$
$AVERAGE(fee)$
FROM S
GROUP BY $hospital,$
$DAY(visit\_date)\ AS\ vdate$

(b) Query B

Fig. 1. Two Query Examples: A and B



(a) Evaluate query A



(b) Evaluate query B from A

Fig. 2. Evaluating Queries A and B

Constructing the shared evaluation plan among multiple queries is necessary to support large-scale queries effectively. Since multiple query optimization (MQO), even on the simple case of aggregating over single columns[7], is NP-complete[21,2,20,16], the global optimization on large-scale queries is impractical. Further, queries generally arrive intermittently in real applications, such as in intelligence monitoring. To make it tractable, a natural solution is incrementally adding new queries to obtain a reasonably good plan within a reasonable time. To do that, the system needs to record the computations of the existing query plan R. Given a new query Q, the system searches the common computations between Q and R, chooses the optimal sharing path, and expand R with new computations to obtain Q's final results.

Assume R contains just query A. Now consider that a new query B arrives. It monitors the daily average fees in a hospital, shown in Fig 1(b). B groups can be obtained by compressing A groups on the $CAT(disease)$ dimension. Further, B's aggregates can be obtained from A as well. Thus the system shares A's results to evaluate B, as shown in Fig 2(b). This sharing process is called *vertical expansion* in this paper.

We implemented two sharing strategies. The first one is to choose the optimal sharing node when there are multiple sharable ones. And the second is *rerouting*. After a new node B is created, the system checks if any existing nodes can be improved by being evaluated from B. These nodes are disconnected from their original parents and connected to B by vertical expansion. If the two queries A and B mentioned above arrive in the reverse order, the sharing can still be achieved by applying rerouting.

The system is built as a part of ARGUS[15,14], a stream processing system atop of a DBMS (Oracle) for immediate practical stream processing functionalities to existing database applications. Besides aggregations, ARGUS also supports selections, joins, set operators, and view references with the incremental evaluation and sharing approaches.

The evaluation study shows that the execution time of incremental aggregation grows much slower than the naive regrouping method as the aggregation size increases, and is up to 10 times faster in our experiments. The study also shows that the incremental sharing provides significant improvement on large-scale queries, and that vertical expansion provides significant improvement on typical data increment sizes.

2 Related Work

Efficient aggregations have been studied widely for data warehousing applications. The CUBE[11], ROLLUP, and GROUPING SETS[2,13,19] generate multiple aggregates by grouping on different sets of aggregating columns (dimensions), and are implemented in commercial DBMS's. These operators allow the query optimizer to share computations among multiple aggregates. [7] proposed a hill-climbing approach to search for efficient shared plans on a large number of GROUPING SETS queries. [22] showed how to choose finer-granularity yet-not-requested aggregates to be shared by multiple aggregates to improve computation efficiency. While these OLAP-oriented approaches consider sharing among multiple aggregates, they assume all queries are available at the time and perform the optimization as an one-shot operation. However, many stream applications, e.g. intelligence monitoring, request registering intermittently arrived queries on an one-by-one basis. Our system accommodates this by recording and searching existing computations to incrementally construct the query plan.

Our work is also relevant to view-based query optimization [9] in terms of searching common computations. However, the optimization only involves identifying the sharable computations without dynamic construction of new computations, and do not concern with incremental aggregations upon data changes. While automatic view maintenance is related to incremental aggregation, previous works[4][12] focus on selection-join queries, not on aggregate queries.

Aggregations over data streams have also been studied. Window aggregates[17] explore stream/query time constraints to detect the aggregating groups that no longer grow and thus avoid unnecessary tuple buffering. Continuously estimating holistic functions, i.e. quantiles and frequent items, in distributed streams [8,18] is another important problem. Since incremental aggregation is not applicable, the approaches focus on approximate techniques and models to bound errors and to minimize communication cost. Orthogonal to these efforts, our work focuses on incremental aggregations for general algebraic functions and support large-scale query systems.

Recent stream processing systems, such as STREAM[3], TelegraphCQ[5], and Aurora[1], are general-purpose Data Stream Management Systems focusing on system-level problems, such as storage, scheduling, approximate techniques, query output requirements, and window join methods. To our knowledge, however, techniques toward incremental aggregation for large-scale queries are not discussed. Our work is orthogonal to these efforts and can be applied as part of the optimizer on such systems.

NiagaraCQ[6] is the closest system to ours. It monitors large-scale continuous queries with incremental query evaluation methods, and registers new queries incrementally with periodical regrouping optimization. However, the system focuses on queries with joins and selections, not on aggregate functions.

3 System Design

In this section, we present the system architecture, incremental aggregation, vertical expansion, and sharing strategies.

3.1 System Architecture and System Catalog

The system builds a shared query network. Each network node, presenting a data stream S, is associated with two tables, the *historical table* S_H to materialize the historical data part, and the *temporary table* S_N to materialize the new data part which will be flushed and appended to the historical table later. At any time, $S = S_H \cup S_N$. An aggregate query may have a GROUPBY clause to specify a set of grouping expressions which is called *dimension set* $\mathbb{D}$. A group value is denoted as $g^{\mathbb{D}}$.

Incremental aggregation on an aggregate node is realized by PL/SQL code blocks instantiated from code templates. The whole query network evaluation is realized by a set of stored procedures that wrap up the code blocks in the order the nodes appear in the network. The stored procedures run periodically on the new data, and incrementally update aggregate results.

Fig 3 and 4 show the system architecture and the system catalog. *System catalog* is a set of system relations to record the topological and node-specific information of the shared query network to allow common computation search, network expansion, and reoptimization. *GroupExprIndex* records all canonicalized group expressions[14], *GroupExprSet* records the dimension sets, and *GroupTopology* records the topological connections of the aggregate nodes. Each dimension set has a *GroupID*, is uniquely associated with an aggregate node in *GroupTopology*, and contains one or more *GroupExpressions* recorded in *GroupExprIndex*. Each node has a unique entry in *GroupTopology* which records the node name, its direct parent (the node from which the results are computed), its original stream table, and the *GroupID*. *GroupColumns* records the projected group expressions and aggregate functions for each node.

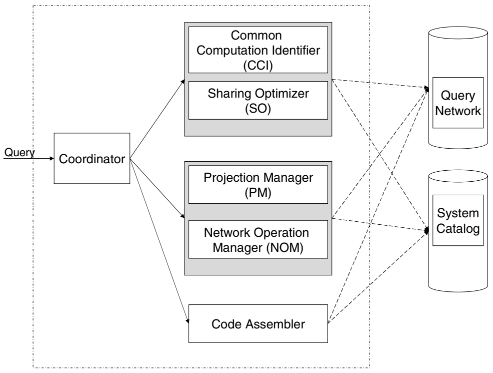


Fig. 3. System Architecture

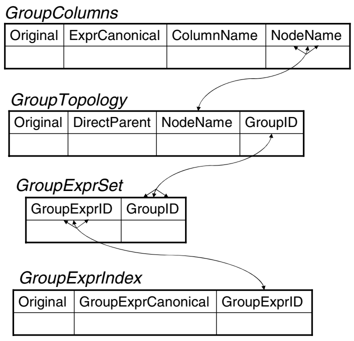


Fig. 4. System Catalog

Given a new query B, the system takes a few steps to expands the existing query network R, so R also evaluates B. First, the common computation identifier identifies the common computations between B and R. Particularly, it identifies all dimension sets $\{\mathbb{D}_A\}$ that are supersets of $\mathbb{D}_B$, identifies the nodes $\{A\}$ associated with $\{\mathbb{D}_A\}$, and then checks whether the nodes in $\{A\}$ contain all columns needed for query B. Second, the sharing optimizer chooses the optimal sharing path, particularly the optimal node A for vertical expansion. Third, the network operation manager and the projection manager expand R and record the changes in the system catalog. And finally, the code assembler assembles code blocks into executable stored procedures.

3.2 Incremental Aggregation

Aggregate functions can be classified into three categories[11]:

Distributive: Aggregate function F is distributive if there is a function G such that $F(S) = G(F(S_j)|j = 1, ..., K)$. COUNT, MIN, MAX, SUM are distributive.

Algebraic: Aggregate function F is algebraic if there is a function G and a multiple-valued function F' such that $F(S) = G(\{F'(S_j)\}|j = 1, ..., K)$. AVERAGE is algebraic with $F'(S_j) = (SUM(S_j), COUNT(S_j))$ and $G(F'(S_j)|j = 1, ..., K) = \frac{\sum F'_1(S_j)}{\sum F'_2(S_j)}$ where F'_i is F''s ith value.

Holistic: Aggregate function F is holistic if there is no constant bound on storage for describing F'. Quantiles are holistic.

Distributive and algebraic functions can be incrementally updated with finite data statistics while holistic functions can not. In this paper, we also refer distributive functions as algebraic, since they are special cases of algebraic functions.

To perform incremental aggregation for arbitrary algebraic functions including user-defined ones, we use two system catalog tables to record the types of the necessary statistics and the updating rules, shown in Tables 1 and 2. Table *AggreBasics* records the necessary bookkeeping statistics for each algebraic function. Argument X in

BasicStatistics indicates the exact match when binding with the actual value while W indicates the wild-card match. Table *AggreRules* records incremental aggregation rules. The rule of a distributive function specifies how the new aggregate is computed from S_H and S_N; and the rule of an algebraic function specifies how the new aggregate is computed from the basic statistics.

Incremental aggregation is shown in Algorithm 1 and is illustrated by Fig 5 on $AVERAGE(fee)$ for query A. Algorithm 1 also shows the time complexity T_i of each step. The merge step is realized with a hash join on A_H and A_N with $T_{hash2} = O(|A_H| + |A_N|)$. If the A_H hash is precomputed and maintained in RAM or on disk with perfect prefetching, the time complexity is $T_{prefetch2} = O(|A_N|)$. The duplicate-drop step is realized by a set difference $A_H - A_N$, which can be achieved by hashing with $T_{hash4} = O(|A_H| + |A_N|)$ or by prefetched hashing with $T_{prefetch4} = O(|A_N|)$. However, we observed that it took $T_{curr4} = O(|A_N| * (|A_N^H|)) = O(|A_N|^2)$ on the DBMS, where $|A_N^H|$ is the number of the groups in S_H to be dropped. If the incremental aggregation is implemented in a DBMS or a DSMS as a built-in operator, both merge and duplicate-drop steps can be achieved with hashing. With the prefetching, the complexity will be linear to $|A_N|$. Therefore, the time complexities on the current implementation, build-in operator with hashing, and with prefetch are following:

$T_{curr} = O(|S_N^2| + |A_H|)$, $T_{built-in} = O(|S_N| + |A_H|)$, and $T_{prefetch} = O(|S_N|)$.

Algorithm 1. ***Incremental Aggregation***

```
0. PredUpdate State. $A_H$ contains update-to-date aggregates on $S_H$.
1. Aggregate $S_N$, and put results into $A_N$. $T_1 = O(|S_N|)$
2. Merge groups in $A_H$ to $A_N$. $T_{hash2} = O(|A_H| + |A_N|)$, $T_{prefetch2} = O(|A_N|)$
3. Compute algebraic aggregates in $A_N$ from basic statistics
   (omitted for distributive functions). $T_3 = O(|A_N|)$
4. Drop duplicates in $A_H$ that have been merged into $A_N$.
   $T_{curr4} = O(|A_N| * |A_N^H|) = O(|A_N|^2)$,
   $T_{hash4} = O(|A_H| + |A_N|)$, $T_{prefetch4} = O(|A_N|)$
5. Insert new results from $A_N$ to $A_H$, preferably after $A_N$ has been sent to the users.
   $T_5 = O(|A_N|)$
```

Fig 6 shows the procedure to instantiate the incremental aggregation code for function $AVERAGE(fee)$ in query A. 1. The function is parsed to obtain the function name and the list of the actual arguments. 2. The basic statistics and updating rules are retrieved. 3. The statistics are parsed and their formal arguments are substituted by the actual arguments. 4. The statistics are renamed and stored in *GroupColumns*. 5. The name mapping is constructed based on above information. 6. The updating rules are instantiated by substituting formal arguments with the renamed columns.

Table 1. AggreRules

AGGREGATE FUNCTION	AGGREGATE CATEGORY	INCREMENTAL AGGREGATION RULE	VERTICAL EXPANSION RULE
AVERAGE	A	$SUMX/COUNTW$	$SUMX/COUNTW$
SUM	D	$SUMX(H) + SUMX(N)$	$SUM(SUMX)$
MEDIAN	H	NULL	NULL
COUNT	D	$COUNTW(H) + COUNTW(N)$	$SUM(COUNTW)$

Table 2. AggreBasics

AGGREGATE FUNCTION	BASIC STATISTICS	BASIC STATID
AVERAGE	$COUNT(W)$	$COUNTW$
AVERAGE	$SUM(X)$	$SUMX$
SUM	$SUM(X)$	$SUMX$
COUNT	$COUNT(W)$	$COUNTW$

3.3 Vertical Expansion and Sharing Strategies

Vertical expansion is not applicable to holistic queries. However, holistic queries can still be shared if they share the same dimension sets. In following discussion, we focus on sharing among distributive and algebraic functions.

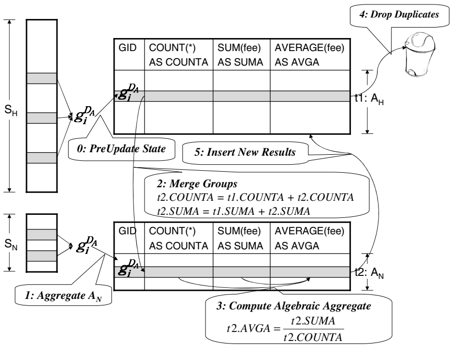


Fig. 5. Incremental Aggregation

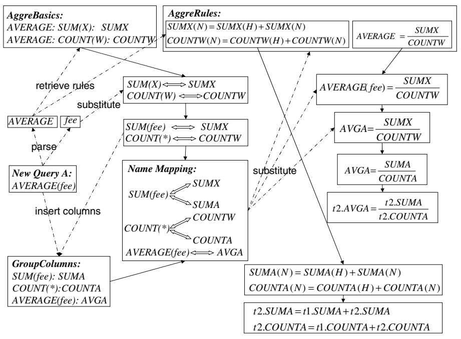


Fig. 6. Incremental Aggregagtion Instantiation

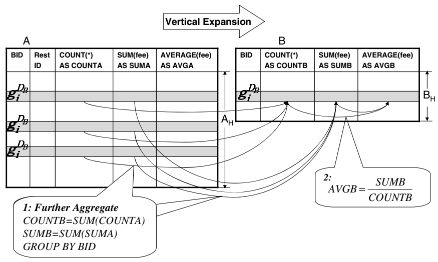


Fig. 7. Vertical Expansion

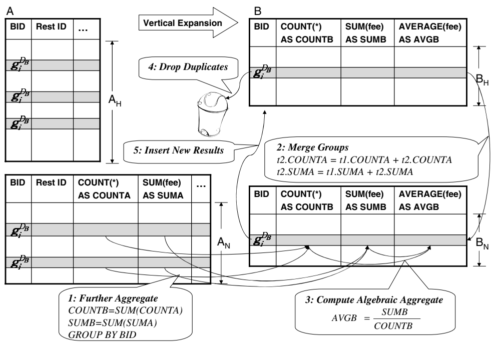


Fig. 8. Incremental Aggregation on Vertical Expansion

Section 1 showed that the query B can be evaluated from the query A (see Fig 2). Fig 7 shows how the vertical expansion creates and initiates B from A. The process is comprised of two steps and takes time of $T^{VInit} = O(|A_H|)$. The further-aggregate step executes the code instantiated from the vertical expansion rules stored in the *AggreRules*, and the algebraic-computing step is applicable only to algebraic functions.

Fig 8 shows the incremental aggregation for $AVERAGE(fee)$ in query B. It is the same to the procedure shown in Fig 5 except the first step which performs further aggregation from A_N instead of from S_N. The time complexities are following:

$T^V_{curr} = O(|A^2_N| + |B_H|)$, $T^V_{built-in} = O(|A_N| + |B_H|)$, and $T^V_{prefetch} = O(|A_N|)$.

Given the new query B, there may be multiple nodes from which a vertical expansion can be performed. According to the time complexity analysis, the optimal choice is the node A such that $|A_H|$ is the smallest. If A does not contain all aggregate functions or bookkeeping statistics needed by query B, a *horizontal expansion* is performed to add necessary aggregate functions to A.

After the new node B is created, the system invokes the rerouting procedure. It checks if any existing node C can be sped up by being evaluated from B. We apply a simple cost model to decide such rerouting nodes. If 1. B contains all the aggregate functions needed by C, and 2. $|B_H| < |P^C_H|$ where P^C is C's current parent node, then C will be rerouted to B. The system applies a simple pruning heuristic. If a node C satisfies both conditions, and a set of nodes $\{C_i\}$ satisfying the first condition are descendants of C, then any node in $\{C_i\}$ should not be rerouted, and so are dropped from consideration.

4 Evaluation Study

We conduct experiments to show: 1. effect of incremental aggregation (*IA*) vs. the naive reaggregation approach (*NIA*), 2. effect of shared incremental aggregation (*SIA*) vs. unshared incremental aggregation (*NS-IA*), 3. performance changes of vertical expansion (*VE*) and non-vertical expansion (*NVE*) with regard to $|S_N|$. The experiments were conducted on an HP PC computer with Pentium(R) 4 CPU 3.00GHz and 1G RAM, running Windows XP.

Two databases are used. One is the synthesized FedWire money transfer transaction database (Fed) with 500000 records. And the other is the Massachusetts hospital patient admission and discharge record database (Med) with 835890 records. Both databases have a single stream with timestamp attributes. To simulate the streams, we take earlier parts of the data as historical data, and simulate the arrivals of new data incrementally. Table 3 shows the historical and new data part sizes used in the experiments.

Table 3. Evaluation Data

	Fed	Med
$\|S_H\|$	300000	600000
$\|S_N\|$	4000	4000

Table 4. Total execution time in seconds

	Fed 350Q	Med 450Q
IA	662	316
NIA	6236	938

Table 5. Vertical expansion statistics

	Pair1	Pair2
$\|S_H\|$	300000	300000
$\|A_H\|$	95050	94895
$\|B_H\|$	10000	10000

We use 350 queries on Fed and 450 queries on Med. These queries are generated systematically. Interesting queries arising from applications are formulated manually. Then more queries are generated by varying the parameters of the seed queries. Some of these queries aggregate on selection and self-join results.

Fig 9 shows the execution times of *IA* and *NIA* on each single query, and the ratio between them, *NIA/IA*. A *NIA/IA* ratio above 1 indicates better *IA* performance. Since there are more fluctuations on diversified Med queries, we show running averages over 20 consecutive queries in Fig 9(b) to make the plot clear. The queries are sorted in the increasing order of $AggreSize = |S_H(A)| * |A_H|$, shown on the second Y axis. $|S_H(A)|$ is the actual aggregation data size of the historical part after possible selections and joins. We experimented with two other metrics, $|S_H(A)|$, $|S_H(A)| + |A_H|$, which show less consistent trends to the time growth and the *NIA/IA* ratio. This indicates that the DBMS aggregation operator on S_H has time complexity of $|S_H(A)| * |A_H|$.

AggreSize indicates the query characteristics. There are about 250 queries in both Fed and Med whose $|S_H(A)|$ is 0, shown to the left of the vertical cut lines and are sorted by the *NIA/IA* ratio. Unsurprisingly, their execution times are very small. The small time fluctuations are caused by the DBMS file caching.

For the remaining queries, incremental aggregation is better than non-incremental aggregation. Particularly, it gains more significant improvements when the aggregation size is large. These large-size queries dominate the execution time in multiple-query systems. Thus significant improvements on such queries are significant to the whole system performance, as shown in Table 4.

Fig 10 shows the total execution times of *SIA* and *NS-IA* by scaling over the number of queries. Clearly, incremental sharing provides improvements, particularly on Fed where queries share more overlap computations.

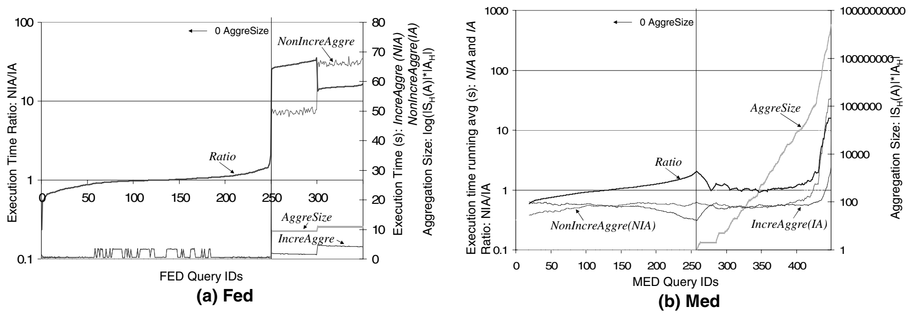


Fig. 9. Sharing

To study how the incremental size $|S_N|$ influences the vertical expansion performance, we select two Fed query pairs and compare vertical expanded plans (*VE*) and non-vertical expanded plans (*NonVE*) on them by varying the incremental sizes from 1 to 30000 tuples in exponential scale, as shown on the X axis in Fig 11. In each query pair, one query B can be shared by vertical expansion from another query A, and their data sizes are shown in Table 5. Fig 11 shows two types of execution times, *IBT* and *ITT*. *IBT* is the time to update the whole incremental batch S_N, indicated on the left Y axis, and *ITT* is the average time to update an individual tuple in each incremental batch, $ITT = IBT/|S_N|$, indicated on the right Y axis. The *VE* performance gets close to *NVE* as $|S_N|$ gets larger, since $|S_N|^2$ in T_{curr} becomes dominant and dims the VE advantage. We expect that this situation can be avoided with the hasing implementation.

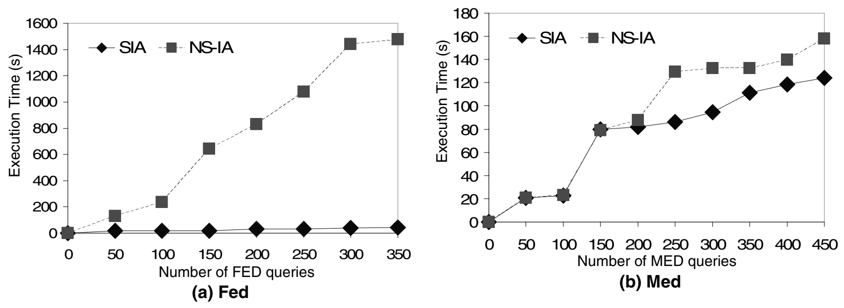


Fig. 10. Sharing

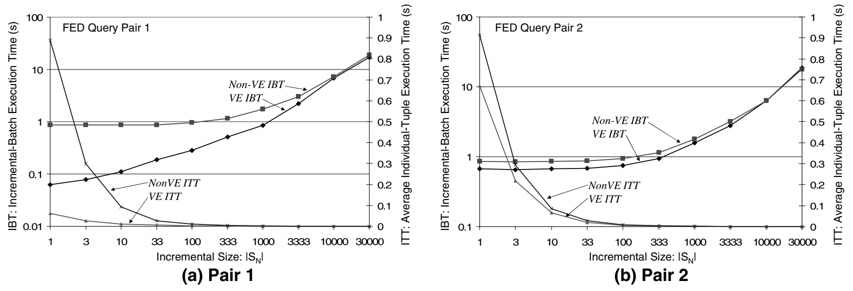


Fig. 11. Vertical expansion

5 Conclusion and Future Work

We implemented a prototype system that supports incremental aggregation and incremental sharing for efficient monitoring of large-scale aggregates over data streams. It also supports incremental aggregation on user-defined functions, which are critical to real-world applications. The evaluation showed that the incremental aggregation improves the performance significantly on dominant queries, and so the incremental sharing over a large number of queries, and that vertical expansion is effective in typical data increment sizes. We plan to migrate the prototype onto an open-source DSMS and implement the algorithms as built-in operators.

Acknowledgements. This work was supported in part by ARDA, NIMD program under contract NMA401-02-C-0033. The views and conclusions are those of the authors, not of the U.S. government or its agencies. We thank Christopher Olston, Phil Hayes, Santosh Ananthraman, Bob Frederking, Eugene Fink, Dwight Dietrich, Ganesh Mani, and Johny Mathew for helpful discussions.

References

1. D. J. Abadi and et al. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
2. S. Agarwal and et al. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.
3. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
4. J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.
5. S. Chandrasekaran and et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, January, 2003.
6. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.
7. Z. Chen and V. R. Narasayya. Efficient computation of multiple group by queries. In *SIGMOD Conference*, pages 263–274, 2005.
8. G. Cormode and et al. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD Conference*, pages 25–36, 2005.
9. D. DeHaan, P.-Å. Larson, and J. Zhou. Stacked indexed views in Microsoft SQL Server. In *SIGMOD Conference*, pages 179–190, 2005.
10. C. Gazen, J. Carbonell, and P. Hayes. Novelty Detection in Data Streams: A Small Step Towards Anticipating Strategic Surprise. In *NIMD PI Meeting*, Washington, DC, 2005.
11. J. Gray and et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.
12. A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.
13. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.
14. C. Jin and J. Carbonell. Toward Incremental Sharing On Continuous Queries. Tech. Report available upon request from authors, Carnegie Mellon Univ, 2005.
15. C. Jin, J. Carbonell, and P. Hayes. ARGUS: Rete + DBMS = Efficient Continuous Profile Matching on Large-Volume Data Streams. In *ISMIS*, pages 142–151, 2005.
16. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.
17. J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conf*, pages 311–322, 2005.
18. C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD Conference*, pages 563–574, 2003.
19. K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB*, pages 116–125, 1997.
20. W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *PODS*, pages 238–248, 1997.
21. T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.
22. M. Zhang, B. Kao, D. W.-L. Cheung, and K. Yip. Mining periodic patterns with gap requirement from sequences. In *SIGMOD Conference*, 2005.