Optimizing Value based tuple retention for stream processing engine (SPE).

Faraz Shaikh
Carnegie Mellon University
School of Computer Science
ISRI - Institute for Software Research International
fshaikh@cs.cmu.edu

Abstract

This work considers deployment of an autonomic storage space management strategy for a very large information processing system. Categories of systems that can benefit from such deployments include storage systems used for data mining and search engines. The common factor binding our target category of information systems is the continual influx of large amounts of data, which needs to be stored persistently for later access. Storage management for such systems often entails having some strategy for retaining the most valued information, while deleting existing information for accommodating new data. Often, storage space for such systems is allocated generously, but eventually all systems have to face a shortage of storage space due to the perennial nature of new data. At times when the storage space is almost full a space reclamation cycle kicks in and tries to free existing data for creating space for new data. One recent approach [1] to retain the most relevant data objects employs retention value functions to assign values to the each data object in the system. The value of the data objects changes over time as they age. A linear programming (LP) approach to optimize the average value of retained data has been described in [2]. Our focus is assessing the application of this LP optimization formulation in the context of stream processing engines (SPE).

The main contribution of this work is a compilation of results describing the performance of an LP formulated cleaning strategy when deployed on real system. The LP cleaning strategy is then compared against other strategies like FIFO and random.

1 Motivation

Large scale information systems have some special needs when it comes to storage management. These management tasks include policy based space reclamation strategies, policy based initial storage assignment, and policy based storage security and guaranteeing service level latency agreements for different classes of data within a system. Most of these management tasks are distinct from the management task requirements of a traditional file store based storage abstraction. The reason for this is that, most files/data stores see relatively less amount of data movements and modifications as data ages. Also, in these systems all information contents are deemed equally important and are treated as first class citizens. Thus file store abstractions do not provide a strong use case for employment of storage management tasks like automatic storage space reclamation. As an example, in a mail/file server every mail/file is important and automatic space reclamation cannot delete old files without the user's input and knowledge.

Information Life Cycle Management (ILM) [3] researches the flow of data from its creation to deletion in a storage system. Numerous techniques and mechanism that fall under the scope ILM include content identification, load balancing, data distribution, backups, availability, security, transparent migrations, hierarchical storage management and disaster recovery. Each of these topics is extensively researched and has resulted in many mature technology solutions [3].

Distributed stream processing systems that store large amounts of data can also use technologies from the ILM landscape. One such focus area is the problem of optimizing the retention of the most valuable data objects, while reclaiming space for new data. The complexity of this problem arises from the following facts

- All data cannot be considered as equal when calculating its value for deletion. Some data classes can be more important and hence more valued than others.
- Reclamation decisions are taken locally in isolation, and the optimization has to be global.
- Interesting decision problems arise when practically implementing a data retention strategy. For example the periodicity of the reclamation cycle needs to be adjusted such that it's not too early or not too late. An early reclamation cycle would mean deleting valuable data before time. On the other hand performing just in time deletions for accommodating new data will induce unnecessary delays in the processing and storage of new relevant data.
- The initial storage assignment problem is constrained by the choice of the features on a storage node. Examples of such constraints are - sensitive data must be assigned to secure storage; legal laws may govern the duration for which some data must be retained (ex. HIPPA) and regularly accessed data must be kept on storage with minimum access latencies.

Borealis [4] is a next generation distributed stream processing engine (SPE). Borealis currently has no load shedding strategy for optimizing retention of data that is stored persistently. Thus borealis provides a good test platform for implementing and testing an algorithm that optimizes the use of persistent storage. It's important to note that Borealis does have a Load Shedding component [5], but this component focuses on active data. Active data refers to the data that is either

entering into the query processing system or is being currently processed within the SPE. Thus the objective of the Borealis Load Shedder component is to improve the latency and throughput seen by the end application, and it does not focus on the value based retention of persistent data objects.

2 Background

Existing SPE load shedding optimizations focus on selective dropping of active incoming packets to clear up processing bottlenecks within a distributed query network. Common load shedding techniques used for this include admission control, and adaptive load distribution. Intermediate results in a SPE are usually stored in main memory for performance reasons. Once processed tuples are published to the monitoring application, the application can decide either to save it persistently or just take note of it. Thus optimizations for persistent storage fall outside the domain of SPE optimizations and are considered to be the responsibility of the end application.

A Linear programming (LP1) approach that takes care of storage node acceptability constraints and optimizes the retention of persistent data is discussed in [2]. The LP formulation is along the lines of the Transportation Problem [6] and models data deletion as flow from the storage nodes to a sink node. The work also describes simulation results, and is intended to be implemented on IBM's 'System S' SPE. The LP1 approach is the first of its kind given that it optimizes retention of "persistent" data in a SPE, a task previously considered to be outside the domain of a SPE.

At this point it’s important to list the different storage requirements of a SPE.

2.1 Storage required for staging

In a SPE aggregate query operators can be specified over a window of tuples. This window can slide based on time or on an absolute number of tuples. The user specifies the size and slide of the window in the query. Aggregates that cannot be calculated without complete knowledge of the window elements viz. joins have to stage intermediate tuples until all the tuples within the window are available. In this case Borealis uses main memory for staging intermediate tuples.

2.2 Storage required for revisions.

Revision processing in a SPE [7] provides a way for correcting erroneous tuples on the input stream. Borealis supports revision processing on user designated input streams. Streams that are expected to receive erroneous tuples are marked in the query diagram. Also, every uncertain tuple on such streams is marked for identification. The uncertain tuples can be corrected within a user specified slack time window. If this tuple correction modifies an already computed result, change notifications are issued to the end application. For the duration of the slack time window the uncertain tuples are stored in main memory.

2.3 Storage required by persistent Connection point (CP) views

A CPView can be applied at any connecting point within the query network. These views store all the tuples at the output of a particular query node. The views can then be used for debugging or supporting ad-hoc queries. Depending on its size a CPView can use a database (the programmers has to take care of it) or main memory to store its tuples.

2.4 Storage required by endpoint applications.

Processed tuples published to the end applications can be persistently saved by the application for future analysis. The storage system used by the application could be either a database or a normal file system. This use case is quite common for ex. analyzing SPE results in a network monitoring application for accountability management. Similarly one might want to correlate previously stored tuple patterns with current patterns for analytics.

Our optimization strategy for retaining the most important data objects can be applied across all the above storage use cases. Application of LP1 on main memory (case 2.1 and 2.2) can help boost the total value of data presented to the SPE but, it would hamper the orthogonal objective of throughput and latency. This is because application of LP1 at internal nodes within the query system may induce significant processing delays within the query network (unless significant buffering and asynchronicity effort is applied), which will affect the latency seen by the end point application. Thus our LP optimization approach is best suited for storage used in cases 2.3 and 2.4.

One other way to design and implement a SPE is to use a Relational database (RDBMS) as a backend and use triggers to forward processed data within the SPE query network. LP1 one can also be applied in this RDBMS centric SPE design. We don't focus on this SPE design paradigm because results show that such implementation is sub-optimal when compared to a dedicated SPE implementation. This conclusion is drawn by eyeballing results from a SPE benchmark study [8].

3 LP Parameter modeling of the Retention problem

The Linear programming formulation of the optimization problem that maximizes value of the retained data has the following parameters.

A. Retention classes (r)

Retention class set r is finite collection of M retention classes each describing important characterizations of the data within the system. For example we can have 2 data retention classes in the systems, one for data that is to be stored on a secure storage and the other for data that is to be stored on normal storage. Each retention class also has a retention value function that describes the value of the data objects of the class as function of their age.

$$r \in \{1,.....,M\}$$

B. Storage Nodes (n)

Storage node set (n) is a set of N storage nodes that are available for tuple storage. Each element in this set describes a single storage node along with its features like capacity, availability, performance, latency etc.

$$n \in \{1,.....,N\}$$

C. Acceptability Matrix (A) of size $M * (N+1)$

The acceptability matrix describes which data retention classes in the set (r) can be assigned to a particular storage node in (n) depending on its features. Ar,n=1 if retention class 'r' can be assigned to storage node 'n'

(A)	0	.....	N
1	1	0	1
..	1	1	0
M	1	1	0

D. Data Movement Cost matrix of size (C) of size $N * N$

The cost matrix C defines the cost of moving tuples from one storage node to the other. Cn,n` is the cost of moving tuples from one storage node (n) to the node (n`). Also Cn,n is 0. Data movement across nodes may be required for load balancing.

(C)	0	.....	N
1	5	0	0
..	1	0	0
N	0	1	0

E. Maximum data movement matrix. (K).

This matrix is similar in dimension to the cost matrix and defines the maximum amount of data that can be moved from node to node during one invocation of cleaning cycle. Knn` is thus the maximum amount of data that is allowed to moved from node (n) to node (n`)

F. Retention values function Vrn(x)

This function gives the cumulative value of data lost when x tuples of retention class 'r' are deleted from a storage node 'n'. V is a function of age of data and shape of the retention curve for a particular data class.

G. The overall maximum data movement allowed in one epoch (k)

This controls the maximum allowed data movement across nodes in a single epoch.

Given the above 8 parameters the optimization problem is solved using network flow techniques on the lines of the Transportation Problem. As an example

let us consider a small deployment of a SPE system with 2 storage nodes N1, N2 and 3 retention classes R1, R2, R3. Additionally for the purpose of modeling a storage node N0 is introduced for describing storage for data that is not currently assigned to any storage node. Storage node N0 can host tuples from any data retention class in the systems. The sample network flow diagram is shown in Fig 1.

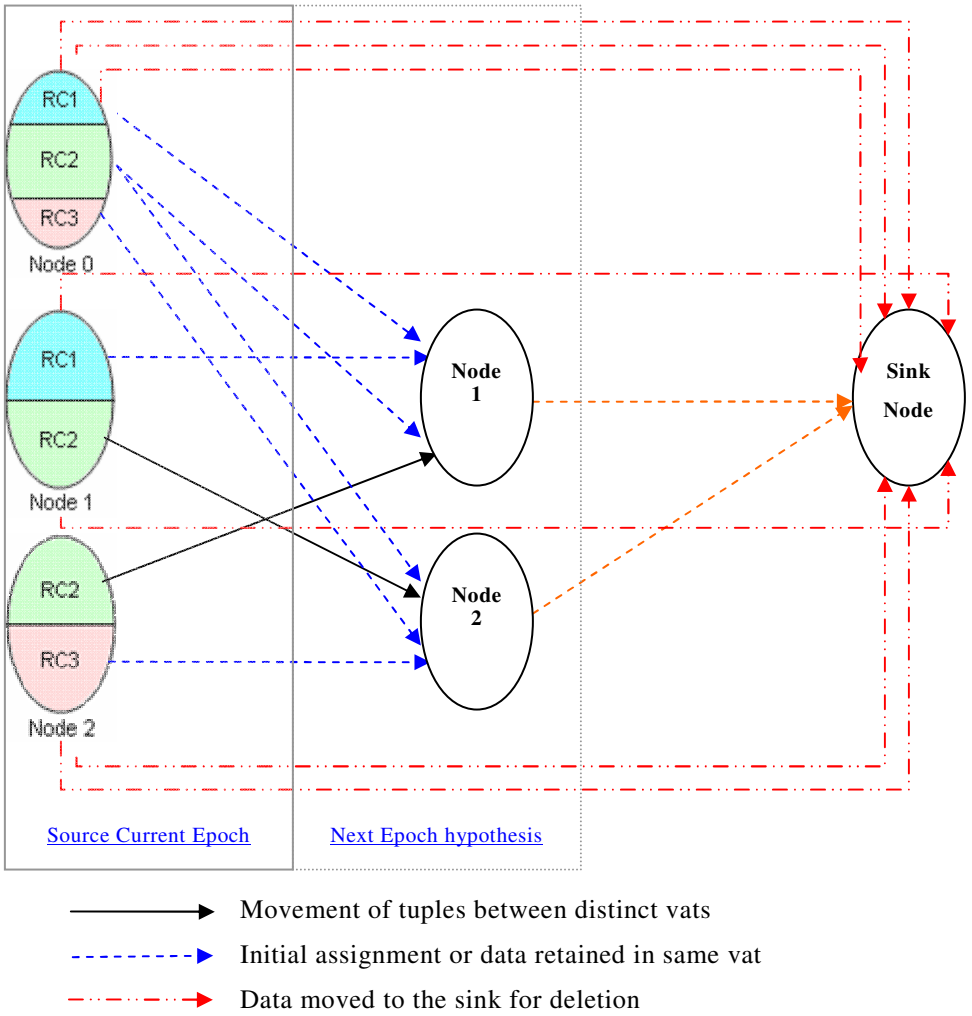


Fig.1.Cleaning cycle data flow for a 2N3R network

3.1 A Sample 2N3R retention problem

A cleaning cycle involves deleting tuples of the least value for accommodating new data tuples. A cleaning cycle is invoked periodically and time between cycles is called an epoch. The diagram in figure 1 describes a 2 node (node1, node2) 3 resource (R1, R2, and R3) retention problem – 2N3R. Resource class RC2 is accepted by storage node1 and node2 while classes RC1 and RC3 are only accepted by storage node1 and node2 respectively. At the end of an epoch storage nodes are full of data (a simplifying assumption for brevity).

In any epoch the storage nodes will host 2 types of record tuples, the data retained from the previous epoch and the estimated data that will arrive during the current epoch. This predicted state of nodes is depicted in the “next epoch

hypothesis" column in fig 1. The current state of storage nodes is represented by source nodes in the "Current source epoch" column of the network diagram. Also, the tuples predicted for an epoch are represented by special source nodes grouped under, Node 0.

The cleaning cycle in Fig 1 can thus be seen as a network in which data flows from current epoch to the next epoch. As a consequence of this data flow some data gets retained (stays in the storage nodes), and some is deleted to make space for new data. All data that is deleted makes it way into the "sink node". The flow along these arcs forms the *decision variables* of the problem. The valid transitions for data from one node to the other are specified via arcs dictated by the acceptance matrix 'A'.

Arcs represent data flow in the network diagram, and all arcs have a cost. In case of our data retention problem this cost is a measure of value lost due to data deletion, or the overhead moving tuples between nodes for load balancing. The optimization goal is to reduce the cost of the data flow and thus minimize the value of the data lost. Hence, if Vrn(x) is function that describes the value lost by deleting x tuples for resource class 'r' from node n, and Cnn' is the cost of moving data between distinct nodes n and n' the optimization function is.

$$Optimize:$$

$$Min\left(\sum_{rv} Vrv(x) \quad + \quad \sum_{nn'} Cnn'\right)$$

3.2 Arcs types and Cost

The arcs in the network can be broadly classified by their cost as follows.

a. Arcs leading into the Sink node.

The flow along these arcs represents deletion of data when the source is existing data in the storage node. The flow can also represent dropping of new data when the source is in the Node0 group. These arcs are marked in red, and their cost is the value of the function Vrn(x). Where 'r' is class of resource, 'n' is the storage node and 'x' is the number of tuples of resource class 'r' deleted from node 'n'.

b. Arcs leading into the nodes of the "next epoch hypothesis" column.

The flow along these arcs represents data to be retained or hosting of new data, the cost along these arcs is zero. These arcs are marked in blue color in Fig 1.

c. Arcs involving data movement across nodes.

The flow along these arcs represents movement of data across distinct nodes for load balancing reasons. For example some record of class 2 might have to be relocated from one node to the other for load balancing. The cost along these arcs is dictated by the cost of data migration.

3.3 Constraints

All constraints in the flow network are "flow conservation" constraints. The sink demands the amount of tuples that need to be claimed in a cleaning cycle. The supply is always greater or equal to the demand because the sink cannot demand more tuples than the sum of the present and predicted tuples. Thus, flow conservation constraints take the form as.

$\because Supply \geq demand$

$in - outflows >= source / demand$

4 Implementation

The infrastructure required of integrating LP1 into an SPE is summarized in the following block diagram. Individual components of the system are discussed next.

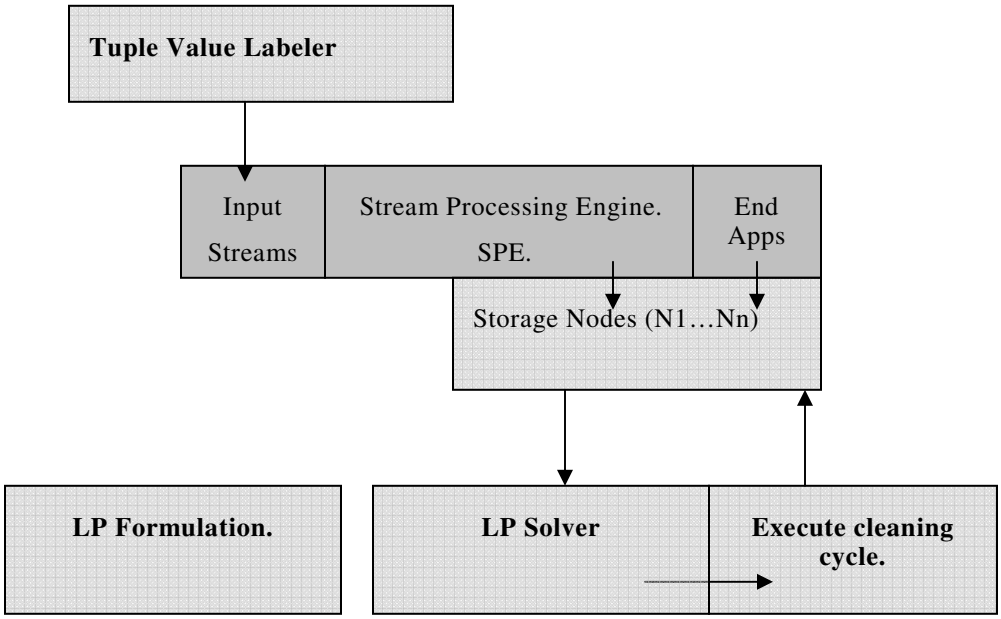


Fig.2. System components

4.1 LP Formulation

This component gathers information for modeling the LP data retention problem. The information includes the number of resource classes, number of storage nodes and the acceptance matrix. For each resource class the parameters of the retention function are also specified. The current retention curve function is

$$V(x) = xc * \left(-\arctan\left((t-p)/a\right) + \pi/2\right)$$

V(x) gives the value of a data object that is 't' time units old. Parameter 'p' decides how long a data tuple of a retention class maintains a stable high value called the 'hold period'. Parameter 'a' decides how steep is decay of the retention

curve called the 'decay period', c is an initial constant value. Parameters P, A, C are defined independently for each resource class. After the decay period the data item continues to hold a very low value as the function tends to 0 as t approaches infinity.

The implementation has fixed model defined in the MathProg [9] language to solve the optimization problem. The parameters to the model can be specified by a separate data file. This separation gives us the ability to keep the model independent as the model parameters change with each epoch.

4.2 Tuple Value Labeler

The tuple value Labeler is a component that adds 1 additional attribute to each input tuple schema. This additional field describes the resource class for the data entering the system. A timestamp field is also added to calculate the age of the tuple. The timestamp attribute need not be added to the input tuple schema and is assigned automatically by the storage node at the time of insertion.

4.3 LP Solver

We have used the GNU Linear Programming kit GPLK [10] solver to solve the network model. The solver provides the optimal number records to be retained from each resource class, and also the number of records to delete/drop that minimizes the value of deleted data. These values are the data flows along the arcs that point into the sink node. Also the solver may provide the amount to tuples to be dropped from new data. These values are the data flow along the arcs that end in sink node and have Node0 group as their source.

4.4 Execute cleaning cycle.

Given the number of tuples to delete for each retention class and storage node combination, the cleaning cycle simply amounts to deleting the specified number of oldest tuples, or dropping specified number of new tuples of a particular resource class. Deletion amounts to firing custom made SQL delete queries of the form.

```
delete from N[n] where ROWID in (select ROWID from N[n] LIMIT [Xnr]) and class = [R];
```

NOTE: This is not the best way to delete 'n' tuples ordered by a field, but our options are restricted by the choice of the version of our DB engine – SQLite. The DB version cannot be upgraded because Borealis and higher version of SQLite conflict in their GCC and libCpp version requirements.

4.5 Storage for Tuples

We have used SQLite as the backend for our SPE applications. Our choice of SQLite [11] as our backend limits us from using elegant forms of delete queries (viz. placing LIMIT clause on DELETE query). In spite of this there are much praises for SQLite than shortcomings. We don't need any esoteric database features for our implementation, apart from the basic create, insert and delete support. Also SQLite is lightweight and this makes it ideal candidate for embedding it large applications.

The control flow is fairly trivial to understand, it involves invocation of the 'cleaning cycle' with inputs describing the estimated number tuples expected to arrive in the next epoch. With this input the parameters for the MathProg model are constructed in a data file. The GPLK solver then solves the optimization problem and provides results to be used by the cleaning cycle.

5 Instrumentation and Results

We summarize the quantitative result of a system using a 2N3R deployment. The tuple labeling is done by adding the class fields to the tuple schema of the input stream. The storage nodes are modeled as distinct database tables. Each cleaning cycle deletes (reclaims space) a fixed percentage of the total tuples in the systems which are then replaced by an equal amount of new tuples. After the cleaning cycle a normalized value of the retained data is collected. Fig3 maps these normalized retained data values across epochs. The results for LP1 retention strategy are compared against results of FIFO, Interleaved and no-cleaning strategies.

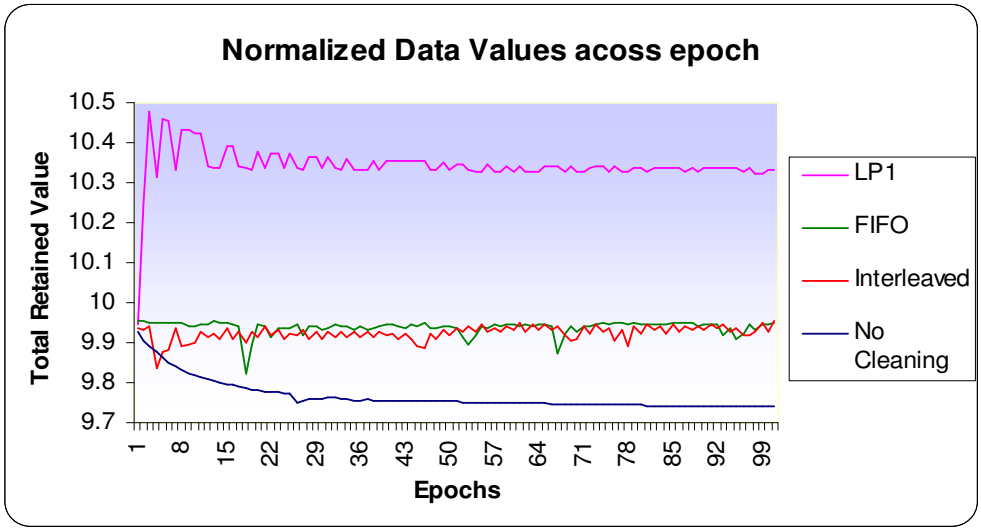


Fig.3. Value retained by various cleaning strategies

Here is a brief explanation of the cleaning strategies shown in fig 3.

I. LP1.

This is the cleaning strategy described in this paper; it solves the retention problem using a linear programming formulation.

II. FIFO

Under the FIFO strategy the oldest tuples for each resource class are deleted to make space for new ones.

III. Interleaved

This is a close approximation to deleting random tuples for making space for new tuples. It randomly deletes odd or even numbered tuples in the order of their arrival.

IV. No cleaning.

Under this strategy no cleaning is performed. The graph for this strategy falls with the epochs because the normalized data retained value factors in (negatively) the total space used by the retention strategy. If this is not done, then the "no cleaning" strategy would always be the best (since everything is retained), but such a comparison won't make any sense.

The 'x' axis plots the time epochs; we measured the normalized retained data value across 100 epochs. The 'y' axis plots the normalized retained data value in the 2N3R system after each epoch. This value is calculated as.

$$retained\ data\ value = \left(\sum_{0-n} V(nth\ tuple) \right) \div n$$

Where n is the total number of tuples in the system. Since each epoch adds same number of tuples as deleted by a cleaning cycle - 'n' is constant, and represents the total number of tuples that can be hosted by all storage nodes. For each such tuple we calculate its value V(tuple) factoring in its age and retention values function of its class (as explained in section 4.1.) For the results 3 resource classes R1, R2, R3 were used. We assigned variable importance to these resource classes in the ascending order i.e. Importance (R1) < Importance (R2) < Importance R3.

The results most certainly conclude that no strategy can perform better than our LP1 formulation of the data retention problem, and that LP1 always produces the most optimum data retention strategy.

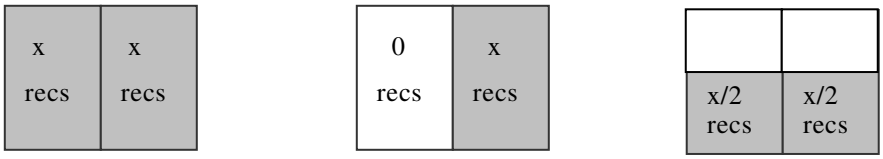
6 Conclusion

LP1 consistently retains more valued data than FIFO and interleaved deletion strategies. Implementing such a system for real deployment is advised, although as the number of resource classes and storage nodes in the system increases the LP formulation has to handle a polynomial increase in the number of constraints. This is not a technological or efficiency problem because many LP solvers exist which can easily solve optimization problems with a very large number of constraints. The only shortcoming is that modeling such optimization problem which is done manually becomes difficult as the complexity of the network flow graph increases. This shortcoming can be overcome by using graphical tools to visualize and construct the network flow diagram and then generate the MathProg model and parameters.

Two subtle corner cases were uncovered when modeling the 2N3R problem. The cases are very briefly summarized next.

6.1 Load balancing problem

Some optimal solution to the retention problem involves deleting X tuples of a single resource class. If there are more than X tuples existing across storage nodes, deletions should be proportionate across these storage nodes Fig3.c. Unfortunately the optimizer suggests greedy deleting tuples Fig3.b taking one node at a time.



(a) Original (b) Greedy deletion (c) Proportionate deletion

Fig. 3 Greedy and proportionate of x number of tuples

Greedy deletion definitely is a problem because it restricts the new tuples to be assigned to only to a few nodes with free space.

6.2 Multiple optimal solutions due to tie between drop and delete decisions

At times in the 2N3R simulation it was clear that ties can occur between the value lost by deleting existing tuples for class 'x' and dropping new tuples of class 'y'. This can result in having multiple optimal solutions to the retention problem that favor either drops or deletes. In case of such ties our implementation always selected the optimal solution that favored dropping new tuples and retaining old tuples (because it was the first solution presented, by the solver). It's important to note that such conflicts can only arise between tuples belonging to *distinct* resource classes. This is because within a single resource class older tuples always have a lower value than newer tuples as new tuples have the smallest age i.e. 0. Depending upon the requirement ties should be broken in favor of either deletion or dropping.

References

[1] DOUGLIS, F., PALMER, J., RICHARDS, E. S., TAO, D., TETZLAFF, W. H., TRACEY, J. M., ND YIN, J. 2004. Position: Short object lifetimes require a delete-optimized storage system. In Proceedings of the 11th ACM SIGOPS European Workshop.

[2] KIRSTEN HILDRUM, FRED DOUGLIS, JOEL L. WOLF and PHILIP S. YU 2008. Storage optimization for large-scale distributed stream-processing systems. In ACM Transactions on Storage Vol. 3, No. 4.

[3] HEWLETT PACKARD. Information lifecycle management technical overview. http://h20427.www2.hp.com/products/storage/cn/zh/pdf/5982-3398EN.pdf (visited Dec 2008)

[4] DANIEL J. ABADI, YANIF AHMAD, MAGDALENA BALAZINSKA, U˘GUR C ETINTEMEL, MITCH CHERNIACK, JEONG-HYON HWANG, WOLFGANG LINDNER, ANURAG S. MASKEY, ALEXANDER RASIN, ESTHER RYVKINA, NESIME TATBUL, YING XING, AND STAN ZDONIK 2005. The design of the Borealis stream processing engine. In Proceedings of the 2005 CIDR Conference

[5] NESIME TATBUL, UGUR CETINTEMEL, STAN ZDONIK 2007.Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In 33rd International Conference on Very Large Data Bases.

[6] DANTZIG G B 1963, "Linear Programming and Extensions" Chapter 3-3. Princeton University Press.

[7] E. RYVKINA, A. S. MASKEY, M. CHERNIACK, S. ZDONIK 2006. Revision Processing in a Stream Processing Engine: A High-Level Design. In 22nd International Conference on Data Engineering (ICDE'06).

[8] ARVIND ARASU, MITCH CHERNIACK, EDUARDO GALVEZ, DAVID MAIER, ANURAG S. MASKEY, ESTHER RYVKINA, MICHAEL STONEBRAKER, RICHARD TIBBETTS 2004. Linear Road: A Stream Data Management Benchmark. In the Proceedings of the 30th VLDB Conference.

[9] GNU MathProg language. http://lpsolve.sourceforge.net/5.5/MathProg.htm (visited Dec 2008)

[10] GNU Linear Programming Kit GPLK. http://www.gnu.org/software/glpk (visited Dec 2008)

[11] SQLite A SQL database engine. http://www.sqlite.org (visited Dec 2008)