B. Tech. Project Report

Shachi Deshpande

Dynamic Scheduling for Maximizing Throughput in Stream Processing Systems

Indian Institute of Technology, Bombay

Supervisor: Prof. S. Sudarshan

Contents

Chapter 1

Introduction

In traditional database design, data is stored on disks and whenever we submit a query to the system, it is optimized and processed on the static data. However now we have stream processing systems that continuously receive tuples of data, and a set of queries has to execute on each incoming tuple. Such systems are required for many important real time applications like stock market analysis, social media news feeds, online advertising, online shopping, etc. Depending on application specific requirements, we need to optimize the query processing w.r.t. different objectives. Such systems have to develop optimized query plans in real time, and take into consideration network related issues like packets of data getting delayed. Such systems have to be resistant to node failures, network congestion, overloading and so on. The queries are typically executed in parallel in such systems, and its important to understand how parallelism is exploited while taking care of objective function. Real time applications like those mentioned above frequently require application of some join and selectivity filters on incoming stream of tuples. Various approaches like batching input tuples, processing tuples out of order, heuristics for dynamic join ordering for sequential execution, etc have been explored. Work done in this area has focused on sequential execution of query done in pipelined execution to take advantage of multiple nodes available for computation. However, parallel execution of query on same tuple on more than one nodes is a valuable addition to these query processing techniques for improving latency of computation. This direction of research has not been sufficiently explored.

In sequential execution of queries, previous work focuses on ordering the execution of constituent filters in a query based on selectivities of filters, capacity of a node to compute results of filter, or combination of both these values [5], [2]. The rationale behind this ordering is to ensure that we minimize redundant computations on tuples which might get dropped somewhere down the query execution pipeline. However these systems can have large latencies if some node is computing results slower than others. Sending tuples to all computing nodes in parallel can improve the latency, but throughput would suffer as the number of redundant computations would be very large. Hence exploring this problem is interesting, especially in environments where the capacities of computing nodes varies dynamically.

Our project looks at streaming query optimization for join queries on star schema of dataset, where a central dataset has to be joined to all other datasets. This central dataset comes from input stream of tuples. Along with pipelined execution, we look at opportunities for processing a given tuple in parallel when the input tuple rate for a server is lower than it can normally handle. We also try to consider the dynamically varying network and server capacity parameters into our dynamic query optimization routine. We aim to improve throughput without blowing up latency wherever possible, as opposed to most of the previous approaches that try to improve throughput. We come up with an algorithm which schedules incoming tuples so that they are processed by all the filters while solving our problem.

The report is further organized into following chapters. Chapter 2 of this report discusses problem statement which we are trying to solve in this project. This is followed by background work done in this area in Chapter 3. In Chapter 4 we discuss the strategy to estimate server capacities using tuple processing times. Chapter 5 explains the different tuple scheduling algorithms to be used in order to solve the problem. Chapter 6 covers the implementation, testing and results. Chapter 7 describes conclusions and future work. This project was done in a group of 2 students, and hence all the following sections of this report are common in our submitted reports.

Acknowledgements :

This project was done in collaboration with *Bikash Chandra*, Ph. D. student and *Deep Modh*, B. Tech. student in Computer Science and Engineering at IIT Bombay.

Chapter 2

Problem Statement

This chapter defines the problem statement for this project. Our goal is to a scenario of maximizing the throughput in Stream processing systems where the capacities of each service may change dynamically and we would like to change in tuple routing policy accordingly in contrast to the earlier work.

2.1 Terminology

Let us consider an input stream S being processed by a stream processing node. For each streaming input tuple, filters $F_1, F_2 ... F_n$ are applied in conjunction. Each filter is processed by a filter service. A filter service runs on one or more remote sites in a distributed setting. To process a filter F_i the corresponding filter service takes some time st_i to process the tuple and has a selectivity of σ_i. Thus after applying the filer, F_i only σ_i fraction of the input tuples need to be processed by the next filter. Our system model is as shown in Figure 2.1. Our goal is to schedule tuples on these filter services for each incoming stream tuples to minimize the latency.

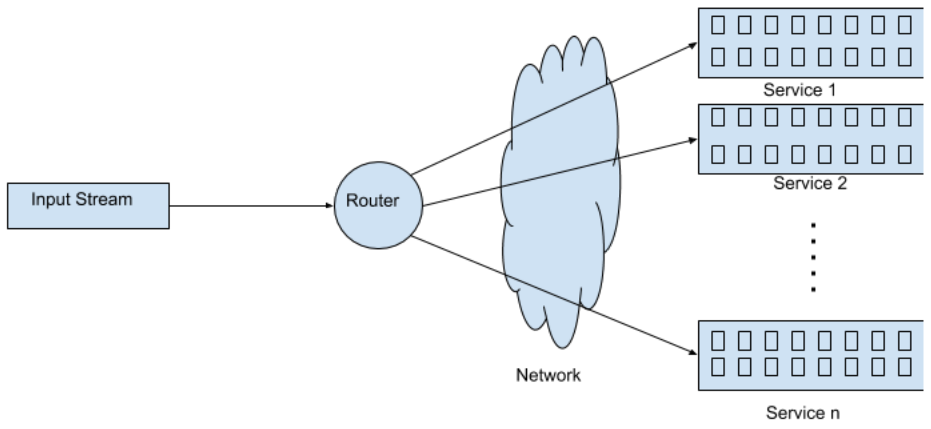

Figure 2.1: router

Each filter service has limited capacity and can process only a limited number c_i of filters per unit time. These filter services could also perform join lookups on a stored relation and perform a filter operation based on the value in the stored relation. The cost of the filter may thus be a combination of disk, network and CPU costs. The cost as well as capacity of the filters may change over time. The node executing the filter may have secondary workloads that reduce the capacity of the filter or if the filter is doing a join the size of the relation may grow over time thus affecting the join cost. Hence the cost and capacity of each filter may need to periodically updated.
We make the following simplifying assumptions. Relaxing these assumptions is part of future work.

1. All filters are applied in conjunction.

2. The filters we consider are such that each streaming tuple can be processed independently of other stream tuples. We do not consider optimization of window operators or aggregates.

3. The cost to apply a particular filter for each tuple is identical.

4. The selectivity of each filter is independent of each other. Hence after applying two filters in succession, with selectivities of σ_1 and σ_2, the output contains $\sigma_1 * \sigma_2$ fraction of the input tuples.

5. Once a filter operator has been started on a tuple, it cannot be preempted.

First we present the solution where there is no dependency among ordering of filters and then we extend it to accommodate the dependencies.

2.2 Service Model

In the service model, we may have multiple relations on a single server or if relation is too large it can reside on multiple servers. We model this cases as a single relation on any given 'service'. Thus if there are multiple relations on a given service then we divide it into more than one virtual services such that each virtual service houses a single relation. Now each service computes a function called 'Filter' over the corresponding relation. This Filter may be a join, select clause or a generic function.

We have assumed that any single join predicate has conditions consisting of attributes in any one relation. We do not consider filters where predicate involves more than one relations.

All the physical servers (behind virtual services) may not stay available as long as our task is going on. Some of the servers might have down-times, and some server can lose connection with our system. Thus if any given server goes down or gets disconnected. then the system won't stop working as long as some physical server still houses the same relation and computes the corresponding filter. Same way, if more physical servers are available, they can be added to the service without interfering our model.

2.3 Optimization Opportunities

If a particular service takes longer time to compute join on a given clause, and input tuples are scheduled such that they visit this service in the beginning, then naturally this slower service would process all the input tuples even though some of them would get dropped at the current or subsequent services after join computation. To avoid this, if the slower service is located downstream w.r.f the order in which join computation is done for a given tuple, then this slower service will receive lower number of input tuples, assuming that some tuples are dropped by the previous services located upstream in the join order. Hence the total time of computing the query result is better in the second arrangement. Thus we would like to determine this order with dynamically changing capacities of services. Network changes also change the response time for the query dynamically. Certain clauses of the join might take more time to compute, depending on the table with which the input stream is to be joined. The rate at which input tuples arrive at our system keeps on changing dynamically, and the number of services in parallel computing join result for given input tuple will change in accordance to this. By parallel computation of join result for a given input tuple, we mean that certain number of services are in parallel computing disjoint sub-parts of the query assigned to them on a given input tuple. This will reduce latency of computation for that tuple, but can potentially reduce the throughput of the system, if the tuple gets dropped at one of the in parallel computing services. Thus the problem of improving throughput without compromising on latency beyond a certain limit in dynamically changing environment is of central interest to our project. Later in the report we explain how to exploit these opportunities.

Chapter 3

Background

There has been considerable prior work in streaming query processing. Frameworks specific to streaming queries have been designed, and algorithms that route tuples dynamically have been proposed.

The idea of ‘Eddy’ was discussed in the work by Avnur et al. [1]. This introduces a query processing mechanism termed as ‘Eddy’, which continuously reorders operators in query plan as it runs. Eddies implement a routing policy for the incoming tuples. They can dynamically change the routing policy while certain queries are being processed, in response to online changes in pattern of input data.

Work done by Condon et al. [3] gives optimal scheduling order to maximize throughput in case of static environment where capacity of each of the server is fixed. Hence this is an important practical setting that this paper considers. This paper presents algorithms for distributed type problem where pipelines run in parallel and throughput maximization is the goal. They view each join operator as having certain rate limit r_i beyond which the operator cannot handle the input tuple rate. They provide an $O(n^2)$ algorithm for ordering the filters and distributing fractions of input tuple flow along various ordering permutations of filter. Consider the case of n filters with selectivities $p_1, p_2 ... p_n$ and rate limits $r_1, r_2 ... r_n$ where $0 < p_i < 1$. The streaming flow is constructed incrementally along the operators in the decreasing order of their rate limits. The residual capacity (rate limit - flow rate) is monitored and the flow is increased till either some operator becomes saturated or residual capacities of two operators become equal. If an operator becomes saturated then the algorithm returns this flow as final result since the flow is optimal else another ordering among operators is computed using residual capacities or each filter. If the rate limits of two or more operators are equal, the operators are grouped into equivalence classes as a single ‘mega-operator’ with the same rate limit. The tuple routing among constituents of the ‘mega-operator’ is done in a way such that the load is equalized.They also prove that under stable conditions, this ordering gives the most optimal distribution of flow rate across all the permutations, when objective is maximizing throughput. They have not looked at the latency aspect, and their algorithm gives the distribution for a given set of values of rate limits on each of the join operators without any special attention given to dynamically changing environment.

Chapter 4

Measurement of Service Capacity

In this chapter, we describe methods to estimate the capacity of a service. We define term '$thread_i$' as number of tuples that i^{th} service can process in parallel. Thus capacity is equal to $thread_i$ divided by the time taken to process a single tuple. We estimate $thread_i$ by keeping a record of tuple return times. The details of this approach are explained below.

4.1 Time required for computing filters

Time taken by next tuple to get acknowledged after computing join = network delay + waiting time at service + service processing time.
Network delay is measured as follows : We periodically monitor the network delay by *pinging* or similar mechanisms for the corresponding service and update the value if change is significant.
Let st_i be service processing time for i^{th} service. We explain how to incorporate wait time and st_i in the next section.

We have to measure both st_i and $thread_i$. For this measurement we do the following

1. We use the precomputed values of st_i and $thread_i$ until a trigger is activated
2. This trigger recomputes the values of st_i and $thread_i$ using feedback tuples
3. A trigger itself might be generated either at regular intervals of time, or following some event, like drastic change in tuple return times.

4.2 Process of Measurement

In this section, we only consider time taken at the service and do not consider network delay, which can be incorporated as mentioned in previous section. Let st_i be the average time taken by i^{th} service. For every service from i = 1 to m, do the following.

1. **Empty the Network**

   Wait until all tuples we have sent so far, are processed and we get the acknowledgement.

2. **Estimate** st_i

   Send one tuple to i^{th} service. Since there is no waiting at the service, time taken by the tuple is the time it takes for one thread to process st_i.

3. **Find an Interval containing** $thread_i$

   The interval of values between which $thread_i$ of a service might lie is found out by an estimation technique which is described in the next part of this section.

4. **Find Number of Threads**

   Use bisection method to figure out the value of $thread_i$.

5. **Capacity**

   The capacity of service is $thread_i$ divided by st_i.

Let $thread_{i,l}$ and $thread_{i,r}$ be left and right boundaries of interval containing $thread_i$ respectively. We want to find $thread_{i,l}$ and $thread_{i,r}$ such that $thread_{i,l} \leq thread_i$ and $thread_i < thread_{i,r}$

Finding an Interval containing Number of Threads:

Initialization :

$thread_{i,l}$ = previously estimated value of $thread_i$ (or 1, at the start of the application)

$thread_{i,r}$ = 2 * $thread_{i,l}$

Iteration Step :

1. Send $thread_{i,l}$ number of tuples (from queue or dummy test tuples) to the i^{th} service to join and let total time taken be $st_{i,l}$.

2. Once all tuples come back, send $thread_{i,r}$ number of tuples to the i^{th} service to join and let total time taken be $st_{i,r}$.

3. Based on the relationship of $thread_i$ with respect to $thread_{i,l}$ and $thread_{i,r}$, determine one of the three possible cases. (Note that $thread_{i,l} < thread_{i,r}$)

Case	Relationship	Observed $st_{i,l}$	Observed $st_{i,r}$
1	$thread_i < thread_{i,l}$	$st_{i,l} \gg t_i$	$st_{i,r} \gg st_i$
2	$thread_{i,l} \leq thread_i < thread_{i,r}$	$st_{i,l} \approx t_i$	$st_{i,r} \gg st_i$
3	$thread_{i,r} \leq thread_i$	$st_{i,l} \approx t_i$	$st_{i,r} \approx st_i$

   When $thread_i$ is greater than or equal to the number of tuples sent, service can process each tuple in parallel and there will not be any waiting on the service. Hence all the tuples are processed in parallel and time required is comparable to st_i. On the other hand, when the number of tuples sent is more than $thread_i$, few tuples need to wait till the processing other tuples finish. Hence time taken will be significantly more than st_i. Thus, based on the value of $st_{i,l}$ and $st_{i,r}$, we can infer the relationship between $thread_i$, $thread_{i,l}$ and $thread_{i,r}$.

4. In case 1, 3; update the values of $thread_{i,l}$ and $thread_{i,r}$ according to the following table and repeat the Iteration Step. In case 2, interval is found.

Case	New $thread_{i,l}$	New $thread_{i,r}$
1	(old $thread_{i,l}$)/2	old $thread_{i,l}$
3	old $thread_{i,r}$	2 * (old $thread_{i,r}$)

We start with a nonempty interval. if the interval contains $thread_i$ then we are done, else in the next iteration we move the interval in the direction of $thread_i$ keeping the boundary of current and next interval same. Thus we do not lose any potential candidate for $thread_i$. Distance between initially chosen interval and $thread_i$ is finite, hence algorithm terminates in finite number of iterations.

Note : In case 2, 3; when time taken by tuples is comparable with st_i, we can update value of st_i by taking average of the two values.

Chapter 5

Scheduling Algorithms

This chapter describes scheduling methods that we considered. The first scheduling method works with a certain fixed routing order which is determined by scheduler. The second scheduling method incorporates the paradigm of work stealing in this scheduling. Work stealing concept is also explained in this section. We build upon these scheduling methods to finally implement a third scheduling method as described in the next chapter. This is considered to be final solution to our problem.

5.1 Scheduler with Fixed Order

For now, we only consider one tuple at a time. Increasing batch sizes can increase the latency of computation, but our algorithms are generalized to any fixed batch size. We can modify batch sizes to improve the performance, and this analysis is part of our future work.

All the requests coming to the application are stored in the *Input Queue(IQ)* in the order of their time of arrival. There are n queues, $Q_1, ..., Q_n$ where i^{th} queue contains the information of the tuples to be processed by i^{th} service. For each tuple, we maintain a bitmap of n bits, where i^{th} bit is 1 if and only if the tuple has been processed by the i^{th} service.

For service i, we define $cost_i$ as the computation cost of filter, $network_i$ as the associated network delay and $thread_i$. Thus, for each service, we maintain the following parameters : $cost_i, network_i, thread_i, t_i$. Note that the $cost_i$ can be calculated by estimating the other parameters, other parameters are estimated as mentioned in previous chapter.

Internals of Scheduler :

Scheduler maintains an array S of size n, containing the information of the services in the sorted order of their cost. Whenever any parameter of any of the service changes, we need to reorder S to maintain the sorted order. Scheduler also has a *Scheduling Queue(SQ)* containing tuples waiting to be scheduled.
A tuple can enter the scheduling queue in two ways.
1. Fetched from IQ : A thread triggers the fetch of tuple when at least one of the Q_i is empty (or size of Q_i is $<$ some constant). Once tuple is being fetched, it is removed form the IQ.
2. When tuple has been processed by one of the services.
Scheduling :
Scheduler pops the first tuple from the SQ say t, and finds the service having the least expected time of processing. Such service say j^{th}, can be found by linear search in S, in increasing order of cost. Here, bitmap associated with t is used to check if tuple has already been processed by a particular service or not.
If no such service exists, then t has been processed by all the services hence return t; otherwise push t in Q_j and update the parameters of j^{th} service. When t arrives back after processing, update the parameters of j^{th} service and set j^{th} bit 1 in bitmap of t. Unless t is dropped(because of join predicates) push t in the SQ.
Each Q_i schedules the processing of the tuples for i^{th} service in First In First Out($FIFO$) scheduling.
Updating Sorting Parameters: Whenever there is significant change in the observed time of computation of filter for tuples, we update trigger the mechanism which updates the $thread_i$ and st_i (time taken by i-th service to process a single tuple).

Updating S : We can update S every time when a tuple is scheduled and when it comes back. Update consists of modifying corresponding service's parameters and recalculating its cost which may result in reordering. Reordering can be done in polynomial time because we only need to find correct position of exactly one element and insert it in array and shift at most n elements.
However it may happen that update on S becomes the bottleneck. To avoid such scenario, we need to ensure that the update time (say U) significantly less than the average time interval between arrival of two tuples at the Scheduling Queue (say δt), i.e. $U << \delta t$ Let F be number of input tuples per second. Let C be average cost of all the services. Since cost is

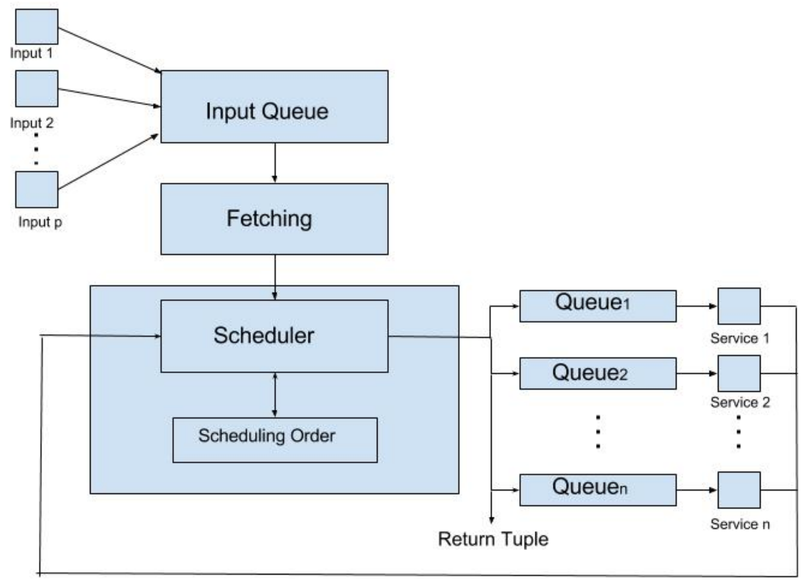

Figure 5.1: Design of Scheduler

time taken, in worst case there will be $1/C$ tuples per second in SQ because of processing on services. Thus $\delta t = \frac{1}{F+\frac{1}{C}}$. We need to ensure $U < \delta t$ at the least. To decrease U; while updating, only update the parameters of the services and do not reorder.

5.2 Scheduler with Work Stealing

Work stealing is a general paradigm of fetching extra work when there is no work left to do at a given worker machine/service. In our system, when a service has low load of incoming input tuples, the scheduler will fetch more tuples from query of other services. Thus these tuples will follow a different route as they are processed by this service before its scheduled order of processing. We believe that this will improve latency since more number of services now process this tuple at a time.

General Approach with Work Stealing: For each tuple probability of survival is maintained, s_i (initially set it as product of selectivity across all services). Also for each tuple maintain number of services that are currently processing the tuple. We term this number as current concurrency level of tuple. Each service fetches the tuples until its capacity is reached.

Tuples are fetched based on current concurrency level of tuple in increasing order. That is, first fetch the tuple having lowest concurrency level (to avoid wastage of resources if tuple is dropped by other services which are currently processing the tuple). If there are more than one tuples having same concurrency level, fetch the tuple having highest s_i(to maximize the chances of survival of the tuple). If there is a tie, we can break it randomly. Bitmap is used to identify if tuple has already been processed by the service or not.

Note that tie occurs when product of selectivity of services which have not processed a tuple is same for two tuples. It can occur when two tuples have been processed by same subset of services in which case there is no metric to differentiate two tuples, which justifies random breaking of tie. Tie can also occur when there are two different subset of services having exactly identical product of selectivity, which is highly unlikely.

If decision cost of selecting the tuple to be fetched is higher than the processing cost of tuple, then it is necessary to take decisions for a chunk of tuples at a time instead of taking decision for every single tuple.

5.3 Final Scheduler with Priority

5.3.1 Setup of Scheduler

Each filter has a *central thread* associated with its queue and there are multiple threads which can process the filter. Central thread maintains the priority queue of the filter and calls other threads to process tuples asynchronously.

Whenever tuple arrives at the scheduler it is added in priority queue of each of the filter with priority 0. Furthermore, at any instance of time, priority of tuple is defined as **numProcessed - (numF $\times$ numCurrThread)**, where numProcessed is number of filters which have already processed the tuple, numF is total number of filters and numCurrThread is number of threads which are currently processing the tuple. Each filter picks the tuple with highest priority to process and removes the tuple from its priority queue after processing it. **Thus, priority queue contains only the tuples which are not processed by the filter.**

We use a priority based work stealing technique to schedule the tuples to be processed by the filters. The number of tuples currently being processed is the more significant component. Tuples that have no filters currently being processed have the highest priority. The (numF $\times$ numCurrThread) component ensures tuples which have processed more filters are given higher priority, thus preventing starving.

With the concept of priority based work stealing, we are able to control the number of filters which could process a tuple at the same time indirectly through the concept of priority. Our priority scheduling algorithm ensures that at low load filters are processed in parallel reducing latency of processing each tuple. At high load running filters in parallel would reduce the latency of some tuples but would lead to longer input queues thereby increasing average latency of processing the filters.Hence filters are processed serially at high loads.

5.3.2 Algorithm

Algorithm 1 Central Thread for Filter

```
Inputs: curFilter: The filter being processed by this thread
     filters : all filters in Scheduler
     priorityQueue : Priority queue associated with filter
     outputQueue: Output of scheduler
     numF: Number of filters to be processed
 1: while true do
 2:     curTuple ← $\phi$
 3:     while curTuple = $\phi$ do
 4:         curTuple ← priorityQueue.getMaximumPriorityTuple()
 5:     end while
 6:     newPriority ← curTuple.priority - numF
 7:     for filter ← filters do
 8:         filter.priorityQueue.updatePriority(curTuple, newPriority)
 9:     end for
10:     result ← curFilter.process(curTuple) // can be processed on multiple threads
11:     curTuple.numProcessed ← curTuple.numProcessed + 1
12:     newPriority ← curTuple.priority + numF + 1
13:     for filter ← filters do
14:         if result = $\phi$ or curTuple.numProcessed = numF then
15:             filter.priorityQueue.removeTuple(curTuple)
16:         else
17:             filter.priorityQueue.updatePriority(curTuple, newPriority)
18:         end if
19:     end for
20:     if curTuple.numProcessed = numF then
21:         outputQueue.add(curTuple)
22:     else
23:         curFilter.removeTuple(curTuple)
24:     end if
25: end while
```

Here we give provide one of characteristic of our scheduler at peak load.

Let Priority = np - N * nc (np = numProcessed, nc = numCurrThread, N = numF)

Claim : At peak load, say t1 is the tuple being fetched by any filter to process, then $nc_{t1} = 0$

Proof : By contradiction. Suppose that $nc_{t1} > 0$. Let t be **any** tuple other than t1. t1 has been picked by filter to process hence priority of t1 is more than priority of t.

$$np_{t1} - N * nc_{t1} > np_t - N * nc_t$$

$$\Rightarrow np_{t1} > np_t + N * (nc_{t1} - nc_t)$$

if nc_t is zero then from above condition $np_{t1} > N$, which is not possible (assumed that $nc_{t1} > 0$). Thus $nc > 0$ for all tuples (including t1).
Which means that in system, every tuple is being processed by at least one filter still system has capacity to process one more tuple. Which contradicts that system is at peak load.

5.3.3 Related Work

We can also compare this approach with the optimal algorithm proposed by Condon et al. [3]. The approach given by Condon et al. [3] discusses the construction of flow of tuples along various permutations of filters while each filter has an upper bound on the rate of incoming tuples it can handle. The flow is constructed by this algorithm in such a way that the throughput achieved is maximal for the given configuration of rate limits and selectivities of filters. Our algorithm does not use any such formal flow construction along permutations of filters explicitly. Instead we use the approach of priority based work stealing to ensure that a tuple can be processed at multiple filters at the same time if some of the filters are idle. This contrasts with the optimal algorithm discussed above in that this optimal algorithm does not allow a single tuple to be processed by multiple filters at once. The optimal algorithm instead allows a given tuple to be serially processed by all the filters one-by-one, and the order of processing would be determined by the flow of which the tuple is a part. Hence we believe that our algorithm will be able to attain better latencies when the work-load is low. Also when the work-load is high, we believe that our algorithm will also converge to serial processing of input tuples. So we use some experiments to see if our algorithm is able to achieve the throughput as obtained by implementing the optimal algorithm, while being able to improve latencies under low and medium loads. As compared to the static scheduler, we have added the concept of priority based work-stealing in this approach. In the following chapter on implementation and testing, we see how our algorithm performs in comparison to the optimal algorithm mentioned above, along with static fixed order scheduling. We see that our algorithm performs better on average latencies but the throughput decreases in comparison to optimal algorithm.

5.3.4 Solution to Dependency Constraint among Filters

Dependency constraint (or precedence constraint) are the constraint among the order of processing the filters, that is among all the filters, some filter needs to be processed before

other. Formally, a Directed Acyclic Graph (DAG) can define the dependency constraint as following : Vertices of DAG is the set of filters. Directed edge u $\rightarrow$ v in DAG is equivalent to the constraint that filter u needs to be processed before filter v for all tuples.
Let *frontier*, associated with a tuple, be set of all filters which can be processed currently for the tuple without violating any dependency constraint. Initially, this set contains filters having no dependencies. In the previous solution without dependency constraint, tuple is added in the queue of all the filters when it arrives, however, in this case, due to constraints, tuple can only be added in the queues of filters in *frontier*.
On a tuple being processed, by say filter u, there is possibility that dependency constraint involving u are satisfied. Formally, for all v such that u $\rightarrow$ v is edge in DAG, it is possible to add v to *frontier* provided all w are processed such that w $\rightarrow$ v is edge in DAG. To add v in *frontier*, we need to check if all such w's have processed the tuple. Checking can be done by maintaining a list of all the filters that are already processed by the tuple. Also, u needs to be removed from *frontier*. Such a solution is possible because dependency graph is DAG.
To optimize checking if the tuple has been processed by all the w's, a counter, say numSatisfied, can be maintained for each filter for each tuple. numSatisfied is the number of w which have already processed the tuple. Thus, whenever u has finished processing, increment numSatisfied for all such v's, and instead of checking at all v's for process status of all w's, check if numSatisfied is equal to number of edges ending at v in DAG.
Tuple can not be added in the queue of other filters otherwise it would violate the dependency constraint. And on the other hand, if tuple is not added to the filters in frontier, then full potential of system is not utilized. Thus, it can be argued that all the properties of our solution without dependency constraint holds true for dependency constraint as well.
Some work regarding this idea of preserving dependency constraints while optimizing throughput was done by Condon et al. [4] Here the prescribed approach is significantly different than the approach used in the case without dependency constraint as discussed in [3]. This is notable because in our solution, dependency constraint is simple extension of previous solution along with book keeping. Also just like [3], solution presented in [4] does not focus on minimizing the latency.

Chapter 6

Implementation and Results

Here we describe the set-up that we used for testing our algorithm. We also describe the results that we got.

6.1 Set-up

We have implemented the scheduling techniques as discussed in the report in Java. The fixed order scheduler and work-stealing based scheduler were implemented as follows. A producer thread generates input stream of tuples at a given rate that can be specified. The scheduler thread receives these input tuples and starts scheduling them according to a certain initial order within the services. According to the feedback of tuple processing times received from the services, this routing schedule is changed dynamically. For fixed order scheduler, this order stays fixed.

Each service is currently implemented as single thread, and hence the capacity translates to CPU capacity. We can extend this to include multiple threads corresponding to given service. Each service thread steals work from other service threads whenever the input queue for that service is filled below certain threshold. For current implementation, this threshold values are hard-coded as certain fractions of total capacity of input queue.

For the current implementation, the filter consists of just joins with HBase tables. We have used a synthetic application to measure the performance of our techniques.The scheduler thread receives these input tuples and schedules them on different HBase nodes using HBase endpoints.

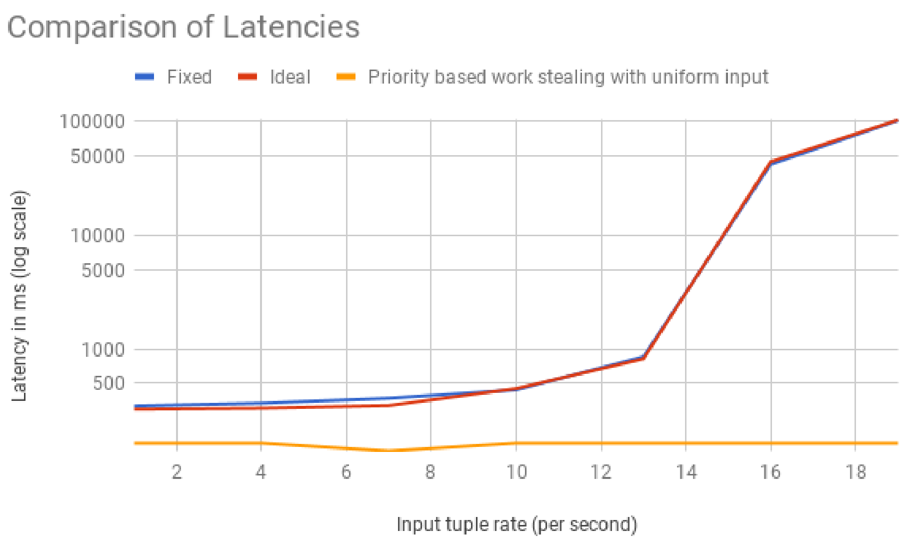

Figure 6.1: Comparison of scheduling approaches by average latencies

We also implemented the algorithm suggested by Condon et al [3]. We implemented this such that it takes the input of rate limits and selectivities of the set of filters and gives output of a scheduling order for each tuple. Each scheduling order has certain probability of being output and this probability calculation is guided by the approach they have discussed in their paper. We test this by adding the scheduling order so outputted by the algorithm to each tuple being routed.

We have tested our implementation on dummy data-set consisting of join with respect to single attribute. We use Hbase for storing data-sets. The input stream as well as the join relations consist of single column family and single column corresponding to each key. We have used a function which simulates the join function itself corresponding to each filter. It internally just sleeps for some time before returning the result. This sleep value can be altered to produce filters of different computation costs. The testing is being done on a quad core CPU with i5 3.1GHz processor. For all the tests, we used three filters on single incoming stream of tuples. Unless otherwise stated, the selectivity of all filters is set to 0.4, and the cost of join is set to 100ms, 150ms and 200 ms respectively. In all the description that follows, ideal approach of scheduling refers to the scheduling approach used in the work by Condon et al. [3]. The fixed scheduling approach refers to using a single fixed ordering of

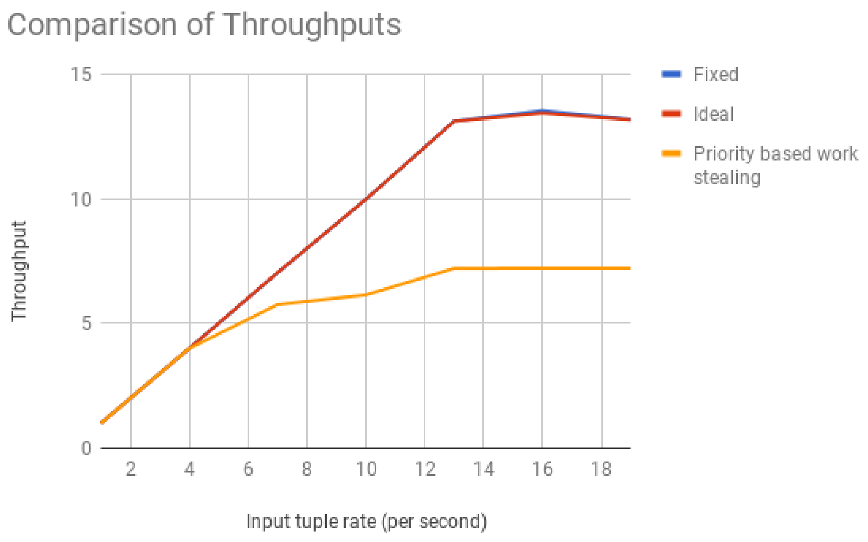

Figure 6.2: Comparison of scheduling approaches by throughputs

filters based on decreasing order of capacities of filters divided by selectivity as heuristic. We compare these two approaches with our approach of scheduling tuples using priority based work stealing.

In the graph 6.1, we see that with increasing input tuple rates, average latency goes on increasing gradually for all the three approaches. However the average latency given by the ideal algorithm by Condon et al. [3] is lesser than that of the fixed order scheduling. We still need to investigate this part further. The latencies given by our algorithm are much better than both the algorithms at all the input tuple rates in the graph. After attaining the maximum possible throughput, the latencies blow up as expected. In graph 6.2 we see that the throughputs given by fixed and ideal algorithm are similar for this configuration of joins, however the throughput given by our algorithm is lesser after some point.

In graph 6.3 we fix the input tuple rate to 20 tuples per second for the priority-based work-stealing approach and increase the cost of all the filters equally, starting with a cost of 10 ms for each of the three filters. We see that when the cost of all the joins was increased, the throughput gradually fell while latencies increased.

In graph 6.4, we again fix the input tuple rate to 20 tuples per second for the priority-based work-stealing approach. We increase the selectivities of all the filters involved in the

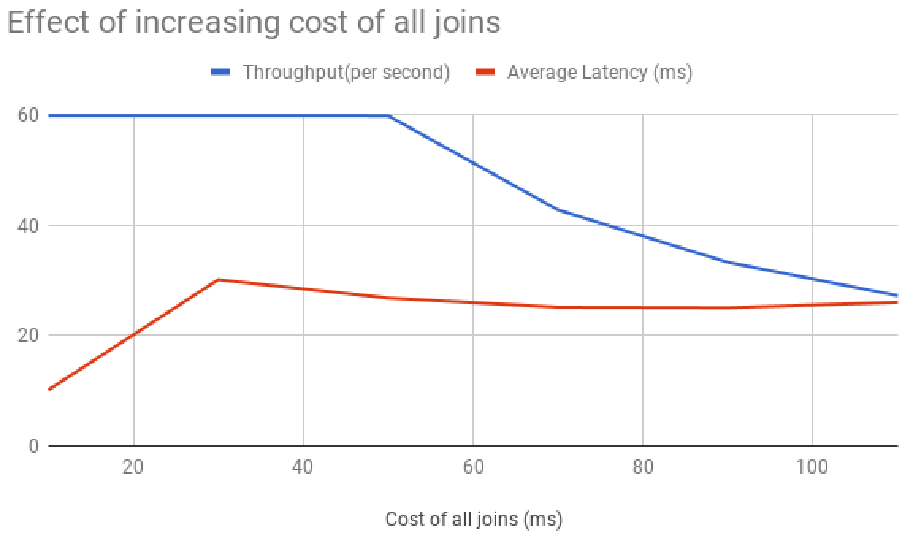

Figure 6.3: Effect of increasing costs of all filters on latency and throughput in priority-based work-stealing approach

computation equally, starting with 0.1 for all the three filters. We see that throughput almost stays same while average latency increases when selectivity increases.

From all these graphs, it is evident that for the current setting, the fixed and ideal scheduling gave almost similar profile of throughputs while the priority-based work-stealing approach gave somewhat lesser throughput. We would like to investigate this further. Similarly, the latencies obtained for priority based work-stealing haven't blown up as expected after a point which needs to be understood better. The latencies of the ideal scheduler are better than the latencies for fixed order scheduler as seen in graphs. The effect of changes in cost and selectivities of filters was also studied for setting of three filters with our priority based work-stealing approach. Increasing costs of filters increases latencies and reduces throughput for this set-up. Increasing selectivities of filters increases latencies while throughput was relatively stable for this set-up. We also checked throughput and latency profiles for poisson distributed input stream for our priority based work-stealing approach. This profile was almost similar to the one obtained with uniform input stream for the same approach, as seen in graph 6.5.

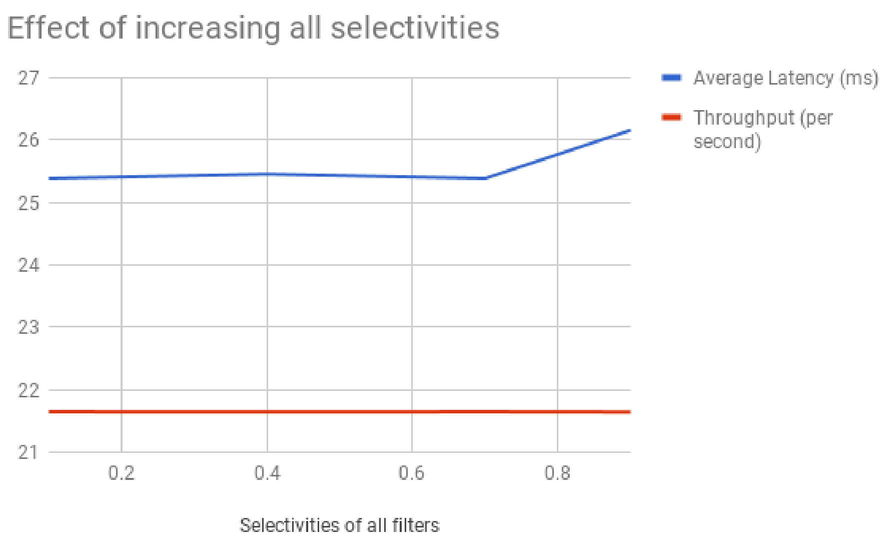

Figure 6.4: Effect of increasing Selectivities of single filter on latency and throughput in priority-based work-stealing approach

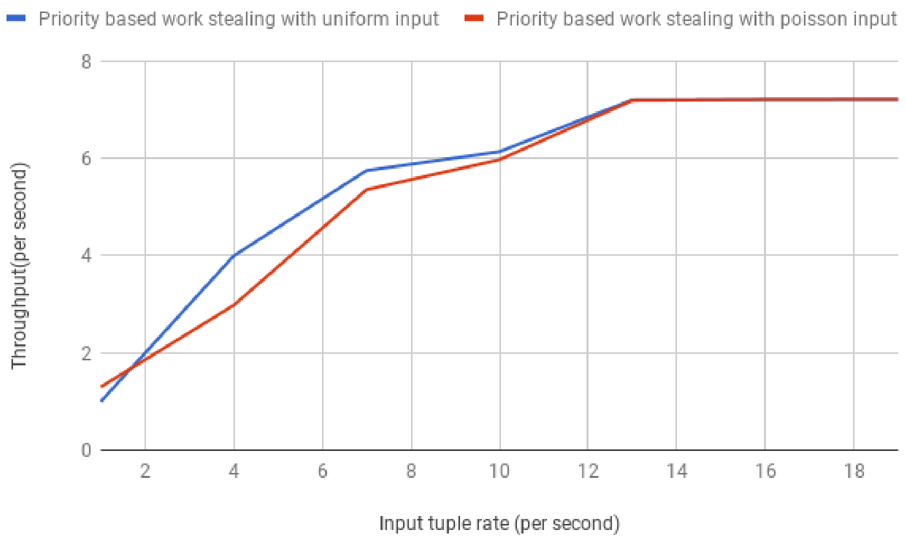

Figure 6.5: Comparison of throughputs with poisson distributed and uniform distributed input stream for priority based work-stealing approach

Chapter 7

Conclusion and Future Work

Our implementation and testing was done on dummy datasets with three filters working on the input stream of tuples. The following conclusions could be made based on these tests.

1. For this setting, the fixed and ideal approaches gave almost similar throughputs though latencies were slightly better for the ideal approach.

2. Changing selectivities of the filters did not affect the throughput significantly though latencies increased. Changing cost of all filters decreased the throughput and increased latencies as expected.

Though we have talked about technique for routing tuples over filters that have dependency constraints amongst themselves, we haven't implemented that part and hence would like to implement it as future work. Also, we need to work on the theoretical aspects of the proof showing how optimal our algorithm is. Specifically, we need to prove it formally that our priority scheduling algorithm ensures that at low load filters are processed in parallel reducing latency of processing each tuple. Also, at high load running filters in parallel would reduce the latency of some tuples but would lead to longer input queues thereby increasing average latency of processing the filters. We have proved some preliminary results but we need to solve this for general case.

Bibliography

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.

[3] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Flow algorithms for two pipelined filter ordering problems. In *Principles of Database Systems*, pages 193–202, 2006.

[4] A. Deshpande and L. Hellerstein. Parallel pipelined filter ordering with precedence constraints. *ACM Trans. Algorithms*, 8(4):41:1–41:38, Oct. 2012.

[5] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 355–366. VLDB Endowment, 2006.