**Seminar Report**

*Deep Modh– 140050002*

# Query Optimization in Database

*Indian Institute of Technology, Bombay*

*Supervisor: Prof. S. Sudarshan*

# Contents

**1 Introduction** **3**

**2 Monitoring Streams using Aurora** **4**
- 2.1 Aurora System Model . . . 4
- 2.2 Aurora Optimization . . . 5
- 2.3 Aurora Run-Time Architecture . . . 5

**3 Runtime Optimization of Join Location** **7**
- 3.1 Framework . . . 8
- 3.2 Modified Ski-Rental Algorithm . . . 8
  - 3.2.1 Recurring Costs After Buying . . . 8
  - 3.2.2 Caching . . . 8
  - 3.2.3 Updates to the Data Store . . . 9
  - 3.2.4 Putting All Together . . . 9
- 3.3 Balancing Computation, Multiple Join, Prefetching . . . 10
  - 3.3.1 Balancing of Computation . . . 10
  - 3.3.2 Multiple Joins . . . 10
  - 3.3.3 Prefetching . . . 10

**4 Parallel and Multi Objective Query Optimization** **11**
- 4.1 Traditional Query Optimizers . . . 11
- 4.2 SCOPE Optimizer . . . 11
- 4.3 Multi Objective Query Optimization . . . 12
- 4.4 MillWheel . . . 12
- 4.5 Orca . . . 12

# Chapter 1

# Introduction

Query optimization is the problem that resides in the very core of any relational database management system. However monitoring systems significantly differ from traditional Database Management Systems. In this seminar report we discuss monitoring streaming in detail with Aurora framework in chapter 2. Chapter 3 contains runtime optimization of join location in parallel Data Management Systems which dynamically balances the load. We conclude the report by brief description of traditional query optimizations, multi-objective query optimization and frameworks like MillWheel, SCOPE, Orca.

# Chapter 2

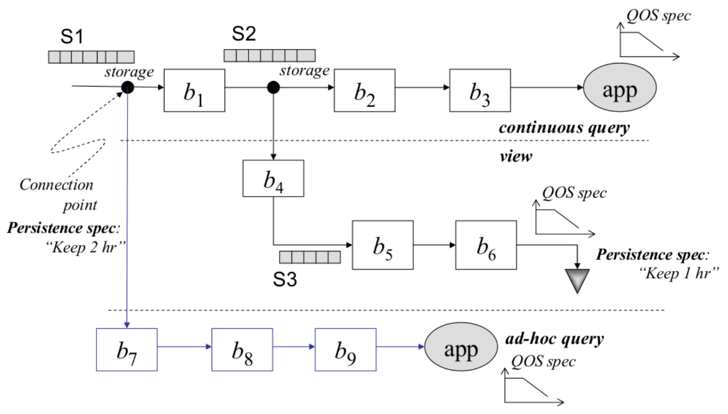
# Monitoring Streams using Aurora

Content and figures of this chapter is based on paper presented by Donald Carney et al.[5].

Monitoring applications differ substantially from traditional DBMS. Traditional DBMS has Human-Active, DBMS-Passive model, whereas monitoring applications has DBMS-Active, Human-Passive model. Secondly, none of the traditional DBMS have implementation that scales to a large number of triggers on the other hand most monitoring applications are trigger-oriented. To handle these challenges Aurora prototype system is designed. In this chapter we first describe the Aurora System Model. In 2.2 we deal with Aurora optimization. We conclude this chapter by discussing Aurora run-time architecture.

## 2.1 Aurora System Model

Aurora is a data-flow system and uses the boxes and arrows paradigm. Tuples flow through loop-free directed graph of processing operations (boxes) and are presented to application as output stream. Aurora consists of two kind of operators; windowed operators like slide, tumble, latch and operators that act on a single tuple (eg. filter).



Aurora supports three modes of operation.

1. Continuous Query :
   Data elements flow into boxes, are processed and flow further downstream. Once an input has worked its way through all reachable paths, it is removed from the network.

2. Ad-hoc Query :
   Connection point (shown as dark circle in figure) is a point that supports dynamic modification to the network and caches flowing items in persistence store for some period of time. Ad-hoc query runs on the information saved at the connection point(s) to which it is connected. Subsequently, it becomes a normal portion of an Aurora network, until it is discarded.

3. View :
   Applications can connect to the end of this path whenever there is a need.

## 2.2 Aurora Optimization

1. Inserting Projections :
   Examine the network and project out all unneeded attributes using filter.

2. Combining Boxes :
   To reduce overhead in execution between of boxes, two boxes can be combined if it leads in lower cost. (eg. combine cheaper operator like filter with any box)

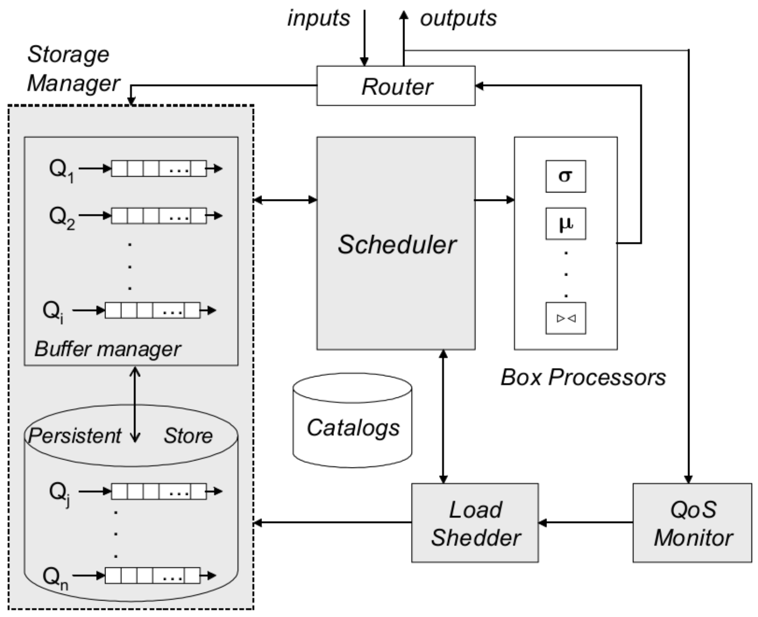
3. Reordering Boxes :
   Interchange two commutative operators if expected execution time is decreased.
   ( condition to interchange can be obtained using statistics which can be used to obtain an optimal ordering )

4. Ad-Hoc Query Optimization :
   Information is organized as B-tree at connection point. Initial boxes can pull information from the B-tree using indexed lookup if possible. (eg. if box is join )

## 2.3 Aurora Run-Time Architecture

Inputs from data sources and outputs from boxes are fed to the router, which forwards them either to external applications or to the storage manager which maintains box queue. Scheduler picks a box for execution and passes it to the multi-threaded box processor. QoS monitor, monitors system performance and activates the load shedder when it detects poor system performance.

Aurora attempts to maximize the perceived QoS provided by application as 2-D normalized QoS vs attribute (eg. delay, tuple drop, value ) graphs. Aurora manages one queue at the output of each box, which is shared by all successor boxes. To allow Aurora to scale up arbitrarily, each queue is stored in disk storage. Blocks in main memory are evicted based on priority.



**Priority assignment** is handled based on expected utility under the current system state or using feedback mechanism which continuously observes the performance of the system and dynamically reassigns priorities to outputs. After selection of individual tuples, to minimize number of I/O operations performed per tuple, tuple trains are constructed. Train scheduling is motivated by avoiding overhead of inter-box bypass and context switching between the boxes.

**Overload** is detected as a result of static or dynamic analysis. Aurora reduces the number of tuples being processed via load shedding. It is assumed that application is coded such that it tolerates missing tuples from data source because of communication failures. Hence Load shedding is not significantly harmful.

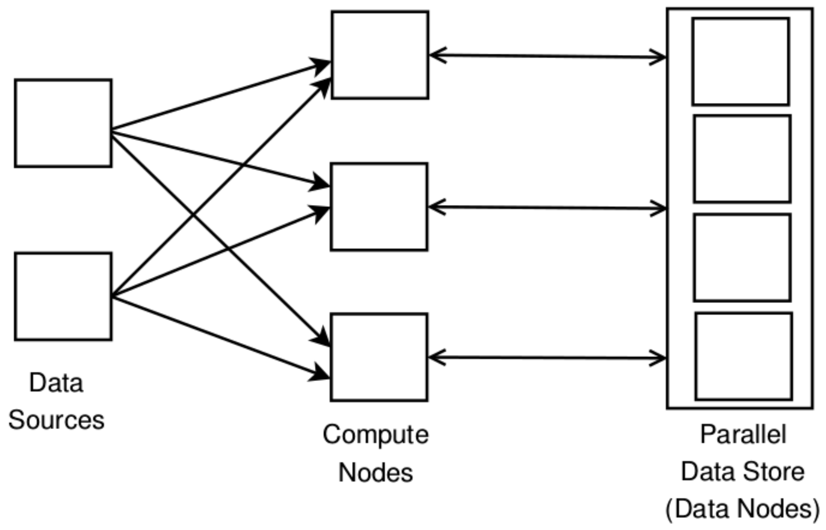
Based on QoS graph, tuples which results in minimum decreases in overall QoS is identified. This interval is converted into a filter predicate which is passed upstream until split-point. If it is difficult to calculate inverse function for upstream box, filter is estimated using trial-and-error which is passed downstream from split point to that box.

# Chapter 3

# Runtime Optimization of Join Location

Content and figures of this chapter is based on paper presented by Bikash Chandra and S. Sudarshan[10].

Parallel systems often need to join a streaming relation with data indexed in a parallel data storage system and compute user defined function (UDF) on the joined tuples. It can be done using reduce side joins (at data node) or map side join (at compute node). However both methods are suboptimal. In this chapter we discuss techniques to make runtime decisions between the two options on a per key basis. First we give framework for the solution. In 3.2 we describe modified ski-rental algorithm for frequency based optimization. We conclude the chapter by discussing multiple joins, prefetching and balancing of computation in section 3.3



## 3.1 Framework

The application need to compute function of the form f (k, p); where “k” is the key of streaming tuple. “p” is a list of parameters to the function. If p is empty then function can return stored value. (computing the join) and does not compute any UDF.

The functions f (k, p) can be invoked in the following ways.

1. Data Request :
   For each key k, fetch the stored value v, compute the function at the compute node.
2. Compute Request :
   Send values (k, p) to the data node, and compute the function at the data node.

The optimization goal is to maximize the throughput(the number of f (k, p) invocations handled in the given time) of the system.

## 3.2 Modified Ski-Rental Algorithm

We can model the decision problem between data request and compute request as classical online ski-rental problem. Compute requests can be considered as renting and fetching the values locally can be considered as buying. Problem is to decide between renting and purchase based on costs. However our problem is significantly different in the following ways, each of which is handled as generalisation to the classical problem.

### 3.2.1 Recurring Costs After Buying

In classical problem, once the item is bought there is no recurring cost, here recurring cost is CPU cost of calculation of UDF at compute node. Let the recurring cost after buying is $b_r$; cost of buying is b; cost of renting is r and m is number of accesses at which we buy.

We should keep renting as long as the cumulative renting cost is less than the cumulative buying cost. Thus, $r * m \leq b + b_r * m \Rightarrow m \leq \frac{b}{r - b_r} \quad \text{if } r > b_r$
else it is always cheaper to rent.

The competitive ratio (total cost/optimal cost) in worst case will be 2 - $\frac{b}{b_r}$.

### 3.2.2 Caching

Classical problem does not take into account limited storage for the items bought, here we have limited storage for the items bought.

Algorithm (on the next page) first checks if key is in memory cache or disk cache. Then based on recurring cost after buying, algorithm decides to make compute request or not (line 11, using recurring cost of memory $b_rM$). condCacheInMemory function determines if the item can be cached in memory (based on availability of cache memory, benifit of key) and thus make data request (line 14). Similarly, based on recurring cost of disk $b_rD$, decision is made. (line 16). updateBenefit and updateCounter are used to to update the caching benefits and access count for the given key.

**Algorithm 1** : skiRentalCaching

```
Inputs: k = data item key
updateBenefit(k)
updateCounter(k)
if k ∈ mCache then
    v ← mCache.get(k)
    localComputeQueue.add(f,k,p,v)
else if k ∈ diskcache then
    v ← dCache.get(k)
    localComputeQueue.add(f,k,p,v)
    condCacheInMemory(k,v,itemSize)
else
    if counter(k) ≤ $b/(r - b_{rM})$ then
        computeQueue.add(f,k,v,p)
    else
        if condCacheInMemory(k,$\phi$,itemSize) then
            dataQueue.add(mCache,f,k,p)
        else if counter(k) ≤ $b/(r - b_{rD})$ then
            computeQueue.add(f,k,v,p)
        else
            dataQueue.add(dCache,f,k,p)
        end if
    end if
end if
```

### 3.2.3 Updates to the Data Store

In ski-rental, items bought are never outdated whereas here tuples may get updated and bought items can no longer be used. To solve this, data node can broadcast to all compute nodes. (or nodes having modified tuples (need to keep track of such nodes) ) This approach can flood the network and result in poor performance. Therefore, with each response to a compute request, the data node also sends the timestamp when the item was last updated. The compute node tracks the timestamp of the last compute request for each data item. If the timestamp gets updated between two compute requests, the counter for the data item is reset.

### 3.2.4 Putting All Together

Let tCompute corresponds to the cost of sending the compute request to the data node, fetching the value of the stored data locally at the data node, computing the function and sending back the computed result to the compute node. Let tFetch corresponds to the cost of sending the data request to the data node, fetching the value of the stored data and sending back the stored value to the compute node. Let tRecMem - cost of computation at the compute node when data is in memory, and tRecDisk - cost of computation at compute node after fetching data from disk. Then,
tCompute $= max(tDiskj, ((s_k + s_p + s_{cv})/netBw_{ij}), tc_j)$
$tFetch = max(tDisk_j, ((s_k + s_v)/netBw_{ij}))$
$tRecMem = tc_i$
$tRecDisk = max(tc_i, tDisk_i)$
Maximum is used because multiple invocations run concurrently and the disk and network access of these overlap with each other.

| | |
|---|---|
| $netBw_i$ | effective throughput of the network in (bytes/sec) at node $i$ |
| $s_v$ | size of stored item $v$ |
| $s_p$ | average size of parameters |
| $s_k$ | average size of size |
| $s_{cv}$ | average size of computed values |
| $tDisk_i$ | average time taken to fetch record from disk at node $i$ |
| $tc_i$ | average time taken to compute the function at node $i$ |

Disk and CPU costs are measured at run-time. Network bandwidth is measured before execution. To accommodate changes to these values, exponential smoothening is done using the following formula, ($\alpha$ is between 0 and 1 )
$value_{t+1} = \alpha value_{measured} + 1$ - $\alpha$ * $value_t$

## 3.3 Balancing Computation, Multiple Join, Prefetching

### 3.3.1 Balancing of Computation

the total time taken is the maximum of the CPU and network time at the compute node and data node is t = $max(compCPU(r_i), compNet(r_i), dataCPU(r_j), dataNet(r_j))$ where CPU is CPU load and Net is Network load at compute node, data node respectively. For the compute node i, suppose the data node selects $r'_{ij}$ requests to be computed at the data node, We substitute $r_i$ with $r_i \;\; r_{ij} + r'_{ij}$ in compute node and $r_j$ with $r_j \;\; r_{ij} + r'_{ij}$ in data node. We use gradient descent to compute the value for $r'_{ij}$ for which the maximum of the four costs is minimum. The value of $r'_{ij}$ determines the number of requests out of total requests that need to be computed at data node j to balance the load

### 3.3.2 Multiple Joins

This approach can be extended without affecting performance by feeding the result of one join as the input of the next join. Optimising join order can be done orthogonally.

### 3.3.3 Prefetching

Requests to the data store are usually blocking. Thus batching and prefetching improves the performance. If the value (data item) from the data node has already been fetched, it can be processed immediately; otherwise the compute node waits till the values become available. Map reduce API is extended by adding a preMap function which can be used to submit prefetch requests.

# Chapter 4

# Parallel and Multi Objective Query Optimization

## 4.1 Traditional Query Optimizers

Content and figures of this chapter is based on PhD thesis of Prasanna Roy[1].

In this chapter, fundamental details related to volcano cascade framework is given. Initially all the equivalent logical plans are generated using a set of transformation rules. Then corresponding physical plans are generated. Both physical and logical plans are represented using directed acyclic graphs where for a particular node, the child nodes represent the results required by the operator corresponding to the current node. We recursively compute costs based on statistics maintained and the representation is space and time efficient. This algorithm of expanding the DAG representation of plan space is proved to be complete. We can add new operators and corresponding transformation rules easily to expand the scope of optimizations. There are certain pruning techniques for efficient search of best plan among all available plans represented in the DAG.

## 4.2 SCOPE Optimizer

Content of this section is based on paper presented by Jingren Zhou et al.[2].

This paper formalizes Data Partitioning and Parallel Plans. Traditionally optimizers adds parallelism once it generates an optimal plan. On the other hand SCOPE combines both these steps. The tuples are partitioned across different machines and result computed by different machine is collected. For partitioning methods like non-deterministic, hash, range partitioning is used, whereas merge sort, concat merge is used for collecting the result. Optimization is done in same way as is done in traditional query optimization, recursively obtain plan using properties for child result and use enforcers as needed.

## 4.3 Multi Objective Query Optimization

Content of this section is based on paper presented by Immanuel Trummerand et al.[3].

The goal of multi-objective query optimization is to find query plans that realize a good compromise between conflicting objectives. In this paper, pareto optimality and pareto frontier is defined. Exact algorithm and approximated algorithm is given. Time complexity analysis of algorithm is included which guarantees worst case time. ( which is not guaranteed by heuristic algorithms )

## 4.4 MillWheel

Content of this section is based on paper presented by Tyler Akidau et al.[6].

MillWheel is a framework for building low-latency data-processing applications that is widely used at Google. A high level overview of MillWheel is given along with some core concepts like Computations, Keys, Streams, Persistent State, Low Watermark, Timers. Overview of architectural details is given. Strong production and weak production is also introduced under Delivery Guarantees and fault tolerance.

## 4.5 Orca

Content of this section is based on paper presented by Mohamed A. Soliman,Lyublena Antova et al.[7].

Orca is a modern top-down query optimizer based on the Cascades optimization framework. Orca can run outside the database system as a stand-alone optimizer. It uses DXL (Data eXchange language) to interact with the database. Orca is multi-core enabled optimizer hence parallelism can be used on distributed system. Orca includes a specialized job scheduler to maximize the performance using parallelism.

Orca optimisation is done in 4 steps.

1. Exploration : Logically equivalent expressions are triggered using transformation rules.
2. Statistics Derivation : Statistics derivation mechanism is triggered to compute statistics in the form of column histograms which are used to estimate cardinality and data skew.
3. Implementation : Physically equivalent expressions are triggered using transformation rules.
4. Optimization : Alternative plan costs are estimated and properties are enforced.

# Bibliography

[1] Rule-Based Query Optimization using the Volcano Framework. Prasan Roy, PhD thesis, IIT Bombay, 2000.

[2] Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. Jingren Zhou, Per-Ake (Paul) Larson, and Ronnie Chaiken in Proc. of the Int'l Conf. on Data Engineering (ICDE), 2010.

[3] Approximation schemes for many-objective query optimization. Immanuel Trummer,Christoph Koch SIGMOD Conference 2014.

[4] Optimizing Join Enumeration in Transformation-based Query. Anil Shanbhag and S. Sudarshan Proceedings of VLDB, 7(12), 1243-1254, 2014.

[5] Monitoring Streams - A New Class of Data Management Applications Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, Stanley B. Zdonik VLDB 2002: 215-226.

[6] MillWheel: Fault-Tolerant Stream Processing at Internet Scale Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle VLDB 2013.

[7] Orca: a modular query optimizer architecture for big data Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia- Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, Rhonda Baldwin SIGMOD Conference 2014.

[8] Efficient and Extensible Algorithms For Multi Query Optimization Prasan Roy, S. Seshadri, S. Sudarshan and Siddhesh Bhobhe Procs. of the ACM SIGMOD Conf. on Management of Data,/May 2000, pages 249-260

[9] Partial Join Order Optimization in the ParAccel Analytic Database Yijou Chen, Richard L. Cole, William J. McKenna, Sergei Perfilov, Aman Sinha, Eugene Szedenits Jr SIGMOD'09

[10] Runtime Optimization of Join Location in Parallel Data Management Systems Bikash Chandra, S. Sudarshan VLDB 2017