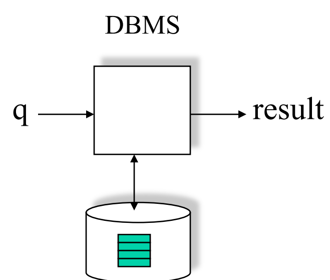
Metis: An Extensible Architecture for Queries over Streams

Alberto Lerner (IBM TJ Watson)

2004

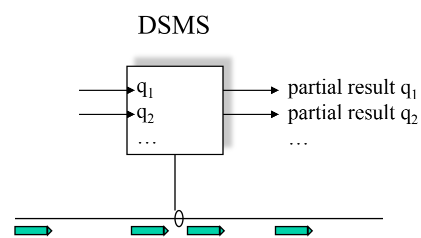
DBMS vs. Data Streams MS

DBMS



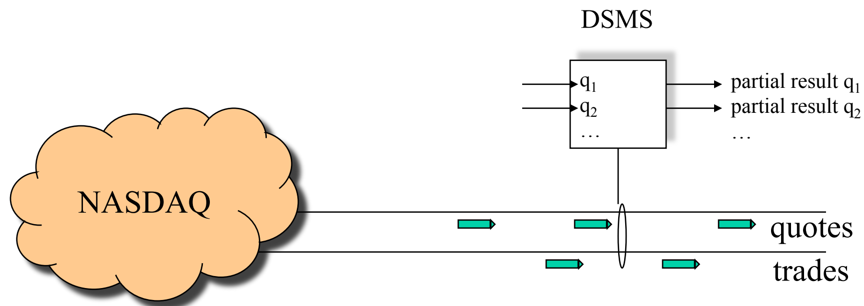
Request-response queries run against a static set of tuples

DSMS



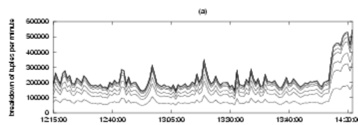
Continual queries run against a dynamic sequence of tuples

Motivating Scenario

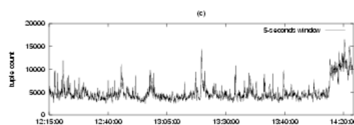


But also sensors, network packets, etc

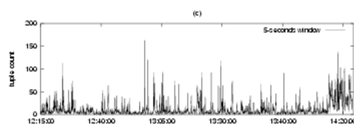
NASDAQ Traffic in Term of Tuples



Combined traffic of the main NASDAQ feeds, 1 min windows

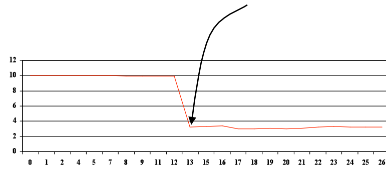


Prime Feed (detailed quotes)
5 second windows

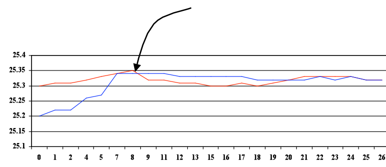


Microsoft Stocks quotes
5 second windows

Typical Queries



"Alert me when IMClone's price drops more than 5 cents between two quotes"

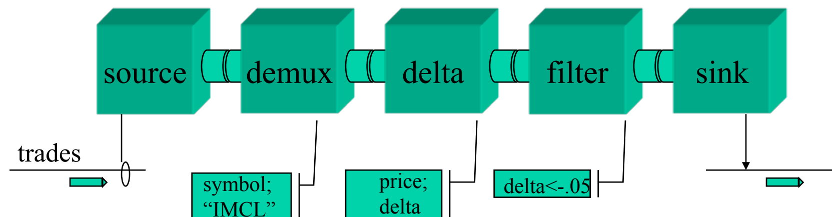


"Alert me when the volume-weighted price of Microsoft is greater than its price"

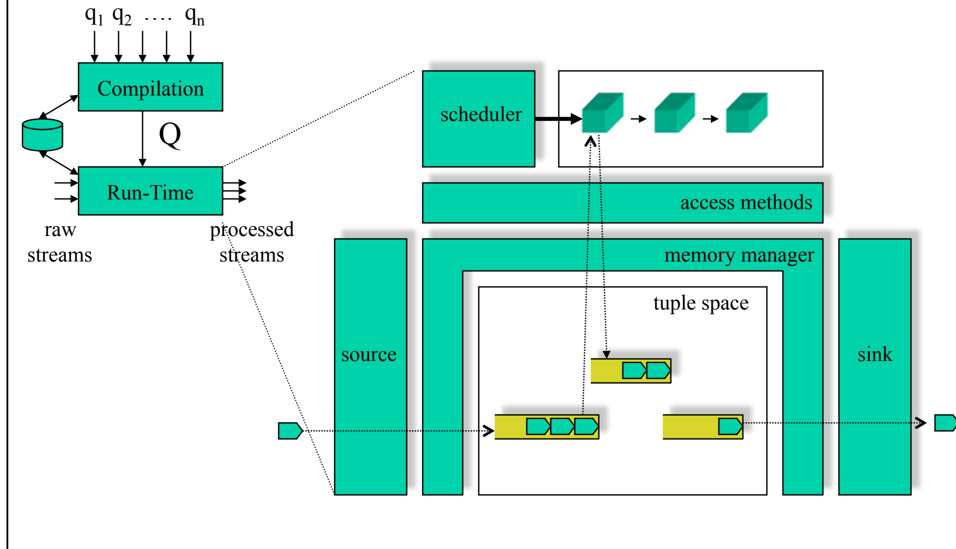
Potentially a very big number of queries, e.g. in a broker or investment bank with dozens or hundreds of traders acting on several markets…

Queries as Composition of Operators

- "Alert me when IMClone's price drops more than 5 cents between two consecutive trades"



Metis's Architecture



Problem: Extensibility

- Different domains may require different sets of operators (one may want to operate on ticks quite differently than one would on video or voice…)
- Designing and implementing new operators has to be easy
- System has support any set of operators; the only requirement is that they communicate through streams

Declaring Operators

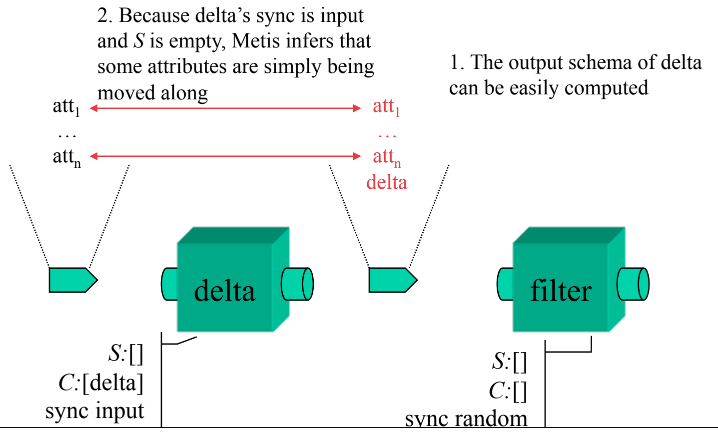
- Metis doesn't need to know specific details about the semantics of each operator
- One just needs to declare an operator's output schema as a function of the input's
- Let I be the list of input attributes; one defines
  - S, the subset of I that the operator suppresses from the input
  - C, the list of attributes that the operator adds to the output
- The list of output attributes O is $I - S + C$ (minus is list difference and plus is concatenation)
- In *delta*, for instance, S is empty and C is the delta attribute

An operator's "Sync"

- Defines an operator's output rate as a function of the input rate, i.e., relative synchronism
  - *Sync input* means an operator outputs one element for each input element read
  - *Sync time* means an operator outputs elements on a clock basis, regardless of the number of elements read
  - *Sync random* means the output rate is unknown or dependent on the input instance
- Rate here is not latency!
- In *delta*, for instance, the output is in sync with the input; that's not the case for *filter*, which is random

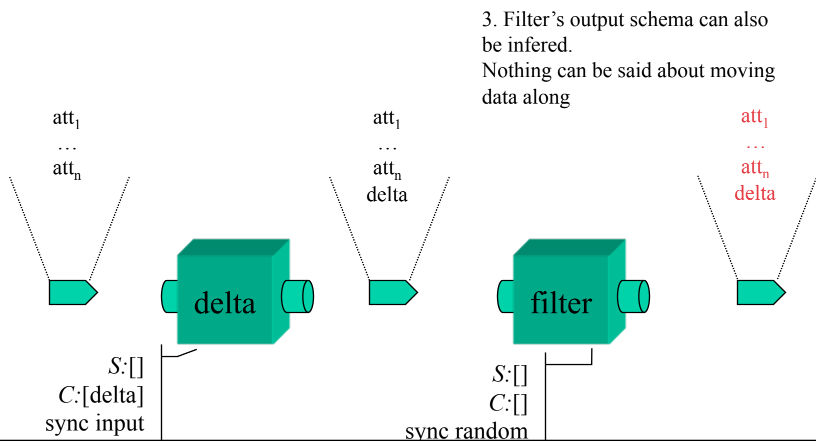
Intermediate Streams' Properties Inference

- Given the operators' definitions, the system can infer important properties about intermediate streams



Intermediate Streams' Properties Inference

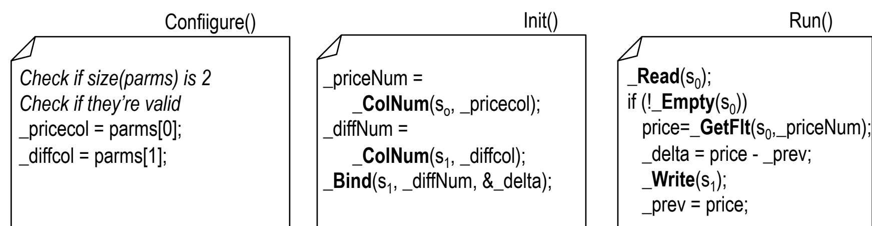
- Given the operators' definitions, the system can infer important properties about intermediate streams

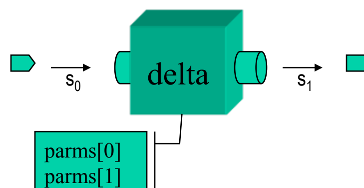


Operator Implementation

- A Metis's operator shouldn't make any assumptions about the layout of streams;
- It counts on a simple yet powerful API to obtain all the information it needs;
- Every operator should inherit from the *Operator* class and should implement the methods
  - *Configure()*, which parses the operator's parameters
  - *Init()*, which prepares the operator to run and
  - *Run(),* which makes the actual data transformation

An Example Operator



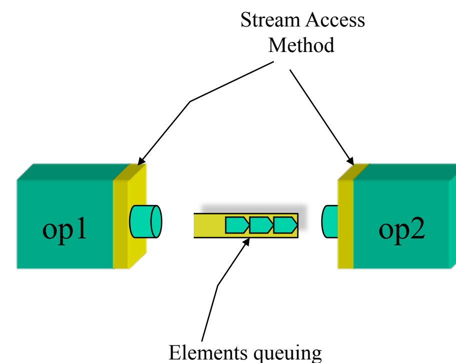


The Case for Queues

- The first impulse is to consider the model a pure pull-data one and call each operator in the sequence the execution graph suggests, but…
  - *Source* is a real-time operator – if not scheduled frequently enough, packets are lost. This means that the calling sequence may be broken so that source could be called
  - Often the output of an operator may be needed by more than one operator. This would force to keep those elements accessible by all the subsequent readers
- … we need queues between (some) operators

Connecting Operators

- Operators simply read and write to input and output streams in a FIFO fashion
- Operators use provided classes such as Stream, StreamSchema, to manipulate data
- The system takes care of the actual queues implementation



Problem: Scaling up

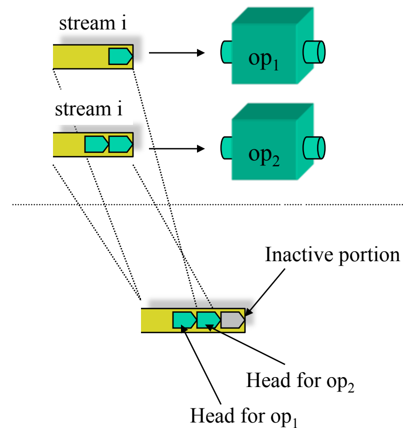
- Queue Management
  - Consider hundreds, thousands of queries; the number of queues between operators may be even bigger
  - Can the space used by those queues be kept to a minimum, even without knowing how big the queues will grow?
  - Can the access to the queues elements be made in a fast way?
- Multi-Query Optimization
  - The more queries we have, the more likelihood of equivalent computations being made in more than one of them
  - Can we detect and eliminate these redundant computations?

Queue Management

- A naïve approach to managing queues storage may yield poor resource usage:
  - Often queues carry very similar data (e.g., delta operator just adds a column). The more data redundancy, the bigger the space needed to process a given workload
  - Queue sizes are highly variable. Errors in memory allocation may cause some queues to run out of space while others keep unused resources.
  - Access to memory is highly influenced by hardware caches (e.g. in Alpha L1=3 cycles, L2=16 cycles, Main=122 cycles). The longer a stream element takes to be accessed, the highest the average latency (the time a stream element takes from source to sink) of the system

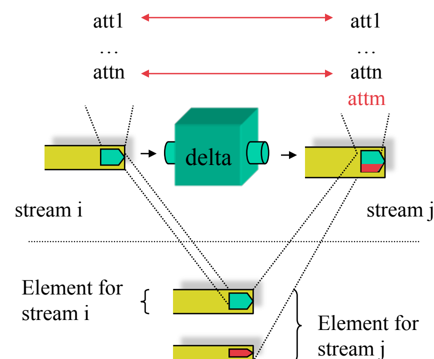
Sources of Redundancy

- Several operators may read from the same queue (e.g., reading the MSFT stream out of a demux operator)
- Queues are actually "logical views" of a lower level structure (we'll talk about it shortly)
- The access method layer provides each operator with its own head of a queue
- When all operators have visited an element, the space it occupies can be recycled
- The queue size reflects the rate difference between the queue writer and its slowest reader



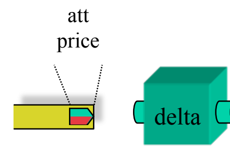
Sources of Redundancy (cont)

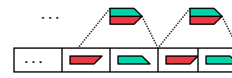
- A given element – or most part of it – may be present in several queues
- Using the same idea that a queue is actually a logical view of an underlying data structure, one can introduce "partial vertical partitioning" to the queues
- Here, several queues may map to a single low-level structure; one low-level structure may be used by several queues

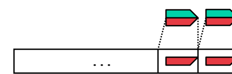


Fostering Cache Efficiency

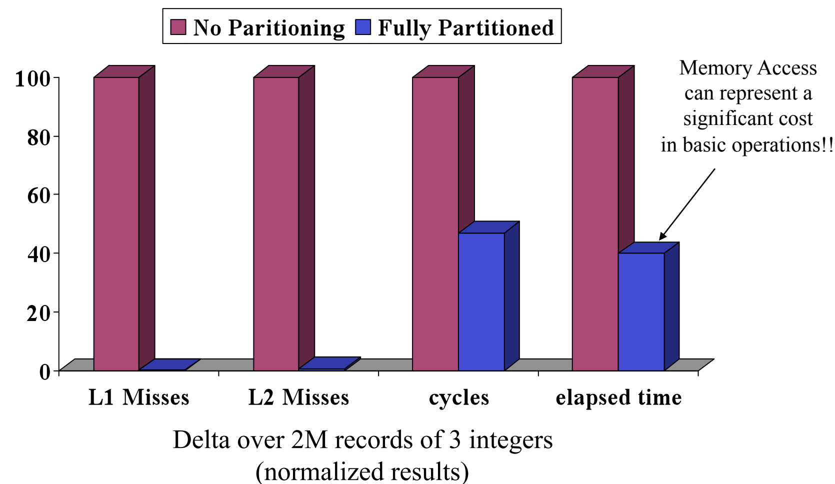
- Operators may move along data they don't touch
- If untouched data is loaded into the caches, it means more misses (for data not used pollutes the cache)
- Thus, partitioning may also be used to separate "touched" from "moved along" data
- Note that filling a cache line requires contiguity







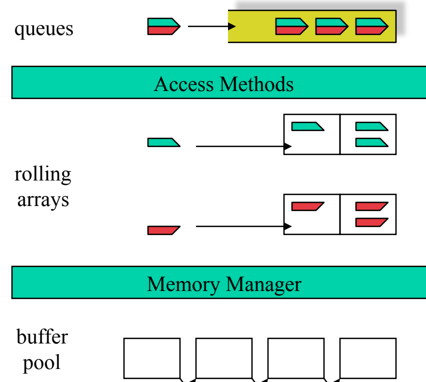
Caching with vertical partition



Delta over 2M records of 3 integers
(normalized results)

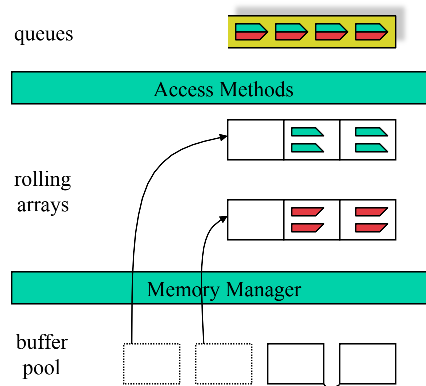
Rolling Arrays

- Underlying data structure that implement queues
- Address both the redundancy and efficiency problems
- Store the **active portion** of a queue
- Memory is assigned to and freed from rolling arrays on demand



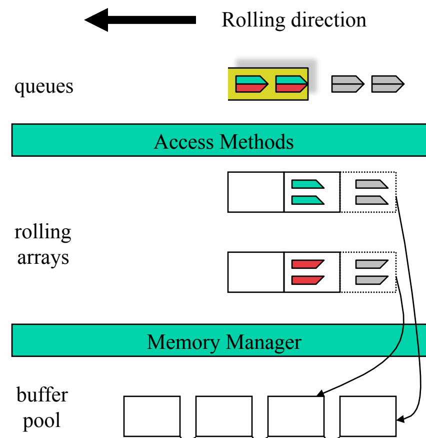
Rolling Arrays

- Underlying data structure that implement queues
- Address both the redundancy and efficiency problems
- Store the **active portion** of a queue
- Memory is assigned to and freed from rolling arrays on demand



Rolling Arrays

- Underlying data structure that implement queues
- Address both the redundancy and efficiency problems
- Store the **active portion** of a queue
- Memory is assigned to and freed from rolling arrays on demand



Mapping Queues to Rolling Arrays

- Given a set of input streams and a set of queries (workload), decide on how to partition each stream
- Fully partitioning increases the overhead of getting a next element (1 get next in a queue needs to be translated to n get next's in rolling arrays)
- An intuition :
  - Traverse the query graph computing for each stream their touched, suppressed, and computed attribute sets
  - Apply heuristics such as
    - Computed columns that are not touched may have space pre-allocated
    - Suppressed data that is not touched may be separated earlier
    - …

Allocating Space to Rolling Arrays

- If memory is assigned in too small chunks to RAs, the lack of contiguity may hurt cache performance
- Too large chunks may cause a stream to hold unused space that could have been used by others
- Adaptive solution intuition:
  - Given the recent growth rate of a stream, decide on how many pages to allocate next
  - The idea is to give a stream space enough to t times worth of recent insertions
  - Consider the time the last request was made and its size s then; if the new request's delta-time is 1/n t times, allocate ceil(n*s) pages.
    - Example: t=1 sec; last request took 1 page; came back in 1/3 of a second -> will give 3 pages

A Performance Metric

- Two variables come to mind that seem appropriate to quantify how well the system behave under heavy workload
  - Memory Usage – a low usage means more data could be processed before load shedding needs to be activated.
  - Overall Cache Misses – a low rate of misses means latency is kept to a minimum
- Note that we want to quantify those variables on a workload that is below the point we would trigger load shedding mechanisms

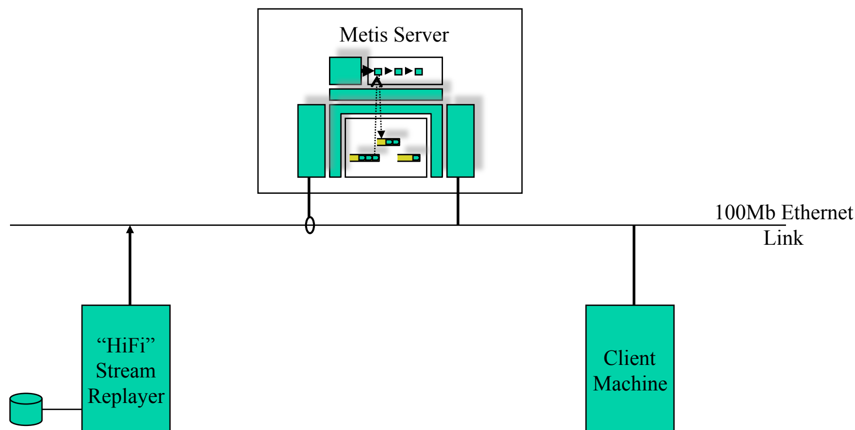
Related Work

- Streaming Systems
  - Gigascope
  - Aurora
  - TelegraphCQ
  - STREAMS
- Cache-Conscious/Main Memory Systems
  - DataMorph
  - Dali/Datablitz
- Other Systems
  - Click Modular Router
  - Staged Databases

Conclusion and Ongoing Work

- Metis makes querying streams as easy as it is querying tables in a relational database
- Metis provides the appropriate support for an extensible set of operators over streams
- It uses information about the layout of the streams and properties of the operators to optimize storage management and query execution
- We're currently working on queue management aspects of the system
- Multi-Query optimization will be next

Demo Setup



Queries

- Q1
  source("eth0") –[trades]>
  sink("client.ibm.com", "9999")
- Q2
  source("eth0") –[trades]>
  demux("symbol", "DELL", "MSFT", "INTC") –[dell, msft, intc]>
  stats("freepages") –[stats]>
  -[dell, msft, intc, stats]> sink("client.ibm.com", "9999", …)
- Q3
  source("eth0", "1") –[trades, packetstats]>
  VWAP("VWAP", "price", "volume") ->
  filter("VWAP>price") –[sell]>
  stats("freepages") –[stats]>
  -[sell, stats, packetstats]> sink("c1.ibm.com", "9999", …)