Data Stream Management Systems
Join Query Operators

Dr. Wenceslao PALMA
wenceslao.palma@ucv.cl

April 2011

Stream

A stream S is an unbounded bag (multiset) of pairs $\langle s, \tau \rangle$, where s is a tuple belonging to S and $\tau \in \mathcal{T}$ is the timestamp that denotes the arrival time of tuple s on stream S.

Query operator

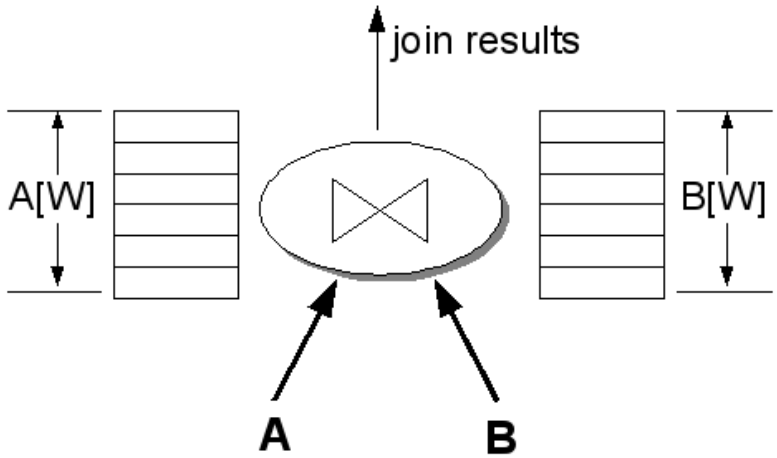
Conceptually, a query operator may be thought of as a function that consumes inputs streams, stores some state, performs computation when new data arrive, modifies the state and outputs results.

The Join query operator

Regarding the Join operator a new implementation is justified by the following reasons.

(a) Since data streams may be infinite, a blocking operator will never see its entire input not being able to produce any result. Solution: implement **streaming symmetric join operators** that processes tuples from the streams in an arbitrary interleaved fashion.

(b) Since data streams are potentially unbounded in size, its is not possible to store join state continuously and match all tuples. Solution: consider a recent portion of the streams based on a **sliding window** that explicitly defines the state of the operator as the set of tuples in the window.



The purge-join-insert algorithm

In general a join operator, implementing sliding windows to limit the size of its states, executes a 3-step process referred to as the purge-join-insert algorithm. For a newly arriving tuple $a \in A$:

(1) a is used to purge tuples stored in window $B[W]$.

(2) a is probed with tuples in $B[W]$ possibly producing join results.

(3) a is inserted in $A[W]$. Symmetric steps are executed for a B tuple.

The three major join operators

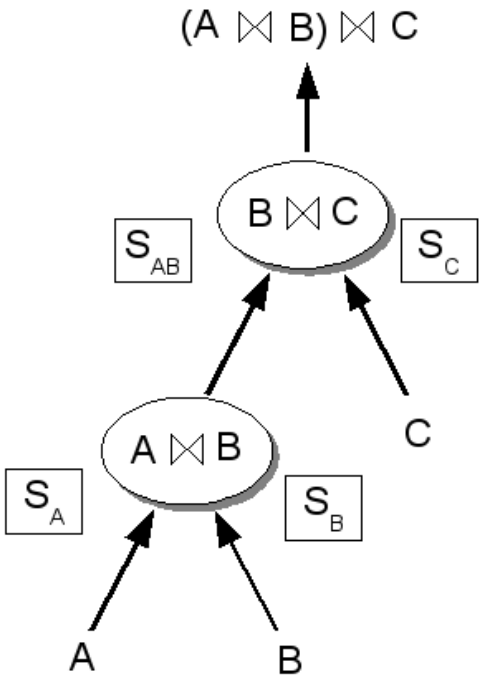
- **BJoin**. Using this operator a query plan is composed of binary join operators that store intermediate results.
- **MJoin**. It takes symmetrically all the streams joining the arriving tuples with the remaining streams in a particular order without storing intermediate results.
- **Eddy**. Using this operator queries are processed without fixed plans. Instead, query execution is conceived as a process of routing tuples through operators where a tuple routing operator adjusts the routing order of tuples on a per-tuple basis.

The BJoin operator

In BJoin, each binary operator keeps two states that stores tuples that the operator has received so far. There are states that store the tuples received directly from a stream, such as state S_A and others states, such as S_{AB} that store intermediate join results. To deal with infinite inputs, the states can be maintained using sliding windows and the join tuples are calculated using the *purge-probe-insert* algorithm.

Example Join Query

```
Select *
From A[range 5min],B[range 5 min],C[range 5 min]
Where A.a=B.a
   and  B.b=C.b
```



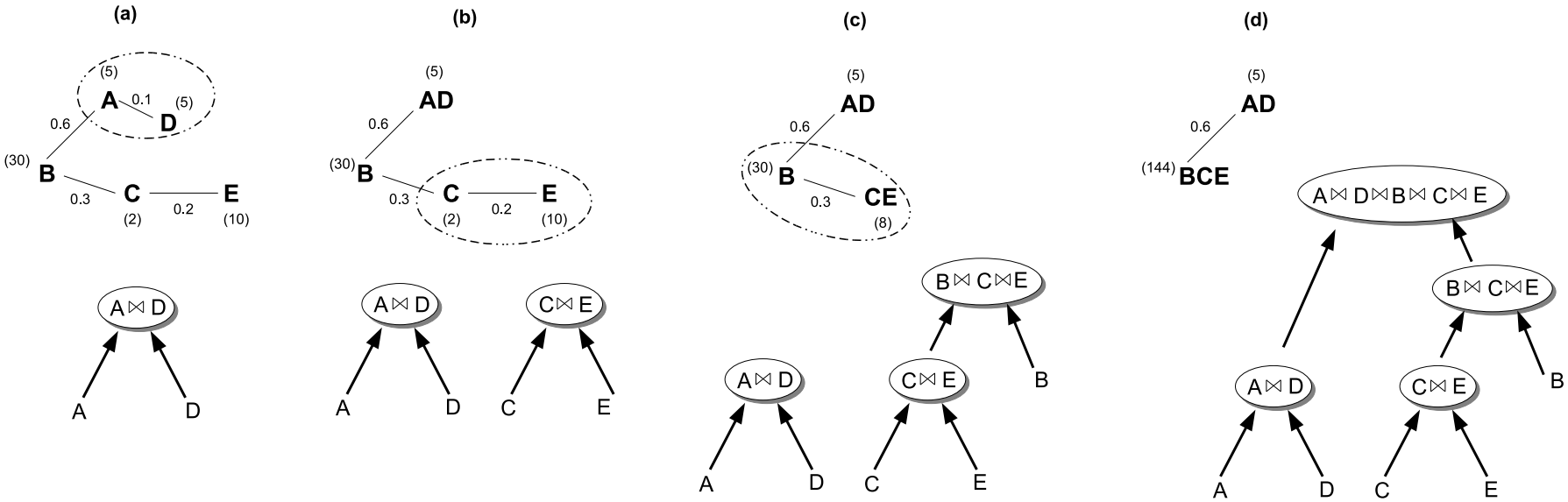
Query Plan

BJoin-Optimization

- The *min-state algorithm* aims at minimizing the total number of intermediate results, thus reducing memory to store them as well as the processing costs (CPU) in generating future join results.
- The *min-state algorithm* is a greedy-based algorithm that computes a BJoin plan in polynomial time.
- The input to the algorithm is a join graph $G = (V, E)$ that represents a multi-join query, where V represents the set of input streams, marked by its stream name V_i and its arrival rate λ_{V_i}, and an edge $(V_i, V_j) \in E$ represents a join predicate between two streams marked by the selectivity $\sigma_{V_i V_j}$ of the join $V_i \bowtie V_j$.
- The algorithm ranks the edges using the following expression $\lambda_{V_i} \lambda_{V_j} \sigma_{V_i V_j}$.

The algorithm ranks the edges using the following expression $\lambda_{V_i}\lambda_{V_j}\sigma_{V_iV_j}$. The smallest value is selected and a join is generated with its vertex. The min-state algorithm does not guarantee to always find an optimal BJoin tree, thus leading the optimizer to be more conservative because it requires more resources than the query actually needs. However, it was chosen for its efficiency i.e. a good plan is found quickly, which is much needed by continuous query processing.

Consider a 5-way join $A \bowtie B \bowtie C \bowtie D \bowtie E$ in Figure (a) with sliding windows of size $W = 1$.



MJoin operator

Mjoin considers n inputs streams symmetrically and by allowing the tuples from the streams to arrive in an arbitrary interleaved fashion. The basic algorithm of MJoin creates as many hash tables (states) as there are join attributes in the query.

Probing sequence

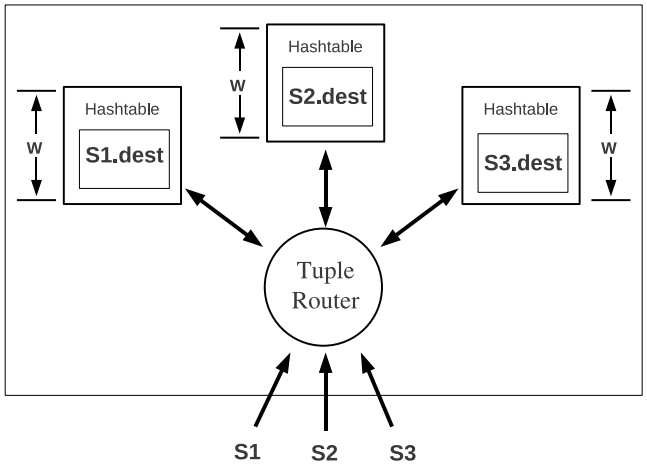
When a new tuple from a stream arrives into the system, it is probed with the other n − 1 streams in some order to find the matches for the tuple. The order in which the streams are probed is called the *probing sequence*.

MJoin operator

Example 3-way join query

Select *
From S1[range 10 min],S2[range 10 min],S3[range 10 min]
Where S1.dest=S2.dest=S3.dest

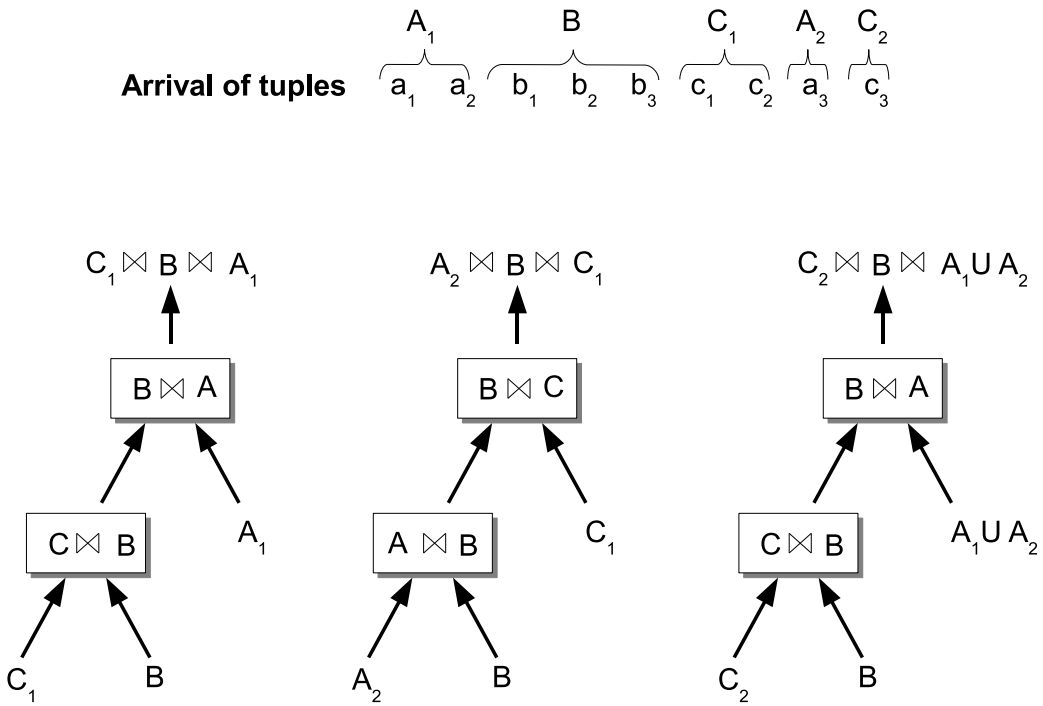
MJoin Operator



Example of Probing sequences

The execution of a MJoin operator can be seen as a sequence of query plans executed using left-deep pipeline plans where the internal state of hash tables is determined solely by the source tuples that have arrived so far.

MJoin operator



Query Plans

MJoin is very attractive when processing continuous queries over data streams because the query plans can be changed by simply changing the probing sequence. **Sliding windows** are adopted to deal with infinite inputs limiting the size of hash tables and the purge-probe-insert algorithm is used to calculate join tuples.

MJoin operator

Drawback

The principal drawback of MJoin is the recomputation of intermediate results because intermediate tuples generated during query execution are not stored for future use. For example, the first query plan generates $(C_1 \bowtie B)$ intermediate results which are not stored and could be used in the query plan $(A_2 \bowtie B) \bowtie C_1$ (see previous figure).

Solution

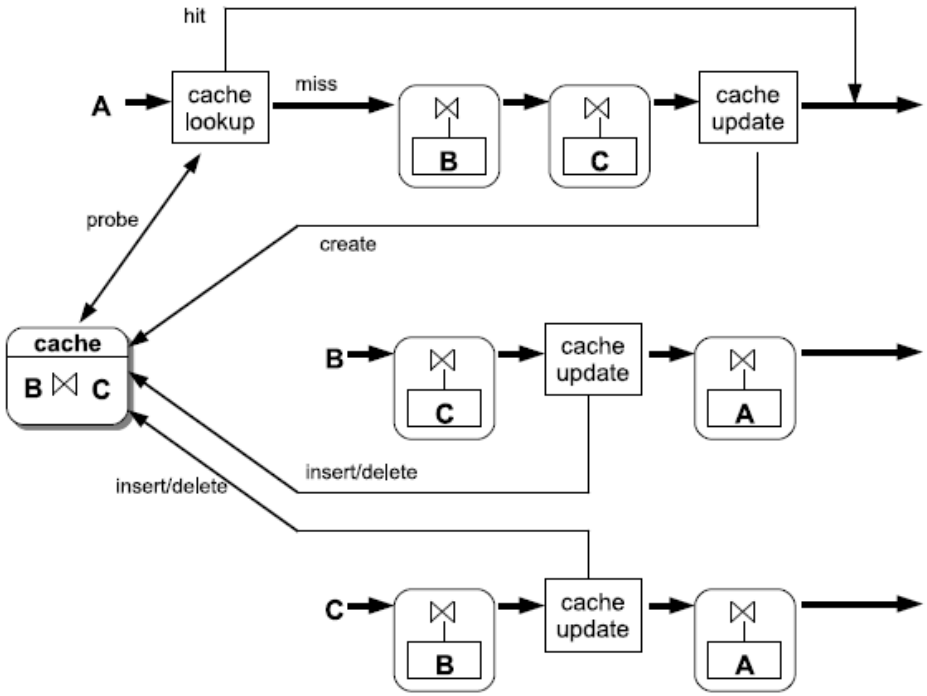
Add/remove temporary caches of intermediate result tuples $\rightarrow$ adaptive caching.

The *A-caching* algorithm

The *A-caching* algorithm improves the performance of MJoin using caches to store intermediate result tuples. In *A-caching* the performance of a continuous join depends on the *probing sequence* and caching. This approach follows a two-step process, to improve the performance of MJoin:

- A *probing sequence* is chosen, independently for each stream, using the *A-Greedy* algorithm.
- For a given *probing sequence*, *A-caching* may decide to add a cache in the middle of the pipeline.

MJoin operator



- When an A tuple arrives, the cache is consulted first to verify if there are results already cached. If so, the *probing sequence* can be avoided.
- If there is a *miss*, the *probing sequence* continues normally and inserts back the computed result into the cache.
- When new B and C tuples arrive, the cache must be updated if the arriving tuples generate $B \bowtie C$ or $C \bowtie B$ intermediate results.

The Eddy operator

Main characteristics

- The Eddy operator is designed to enable fine-grained runtime control and adaptively approach the optimal order of join operations at runtime.
- Queries are executed without fixed plans.
- An optimized join order for each incoming tuple is computed individually.

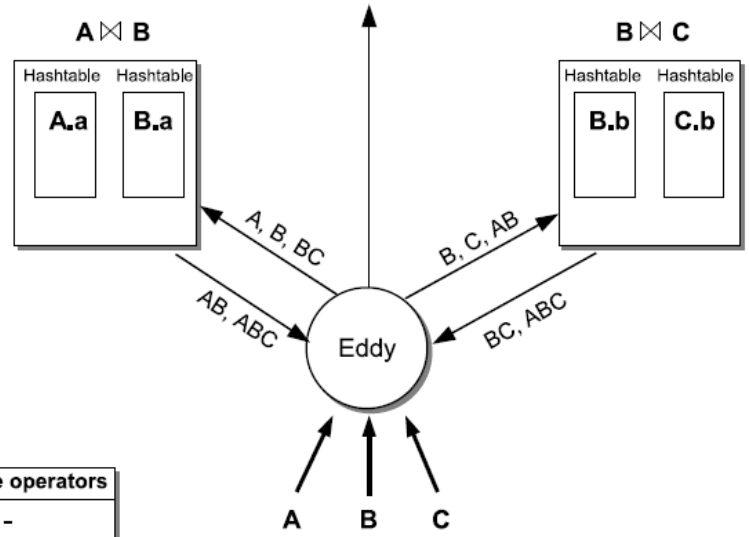
How Eddy works

- An Eddy is a tuple routing operator between data sources and query operators as joins.
- It monitors the execution and makes routing decisions by sending tuples from the data sources through query operators.
- As a result, the routing destinations for tuples alone determine the query plans executed by the Eddy.

The Eddy operator

Example Join Query

Select *
From A, B, C
Where A.a=B.a
and B.b=C.b



Routing Table

Base tables	Routed through	Eligible operators
A	A ⋈ B	-
A,B	A ⋈ B	B ⋈ C
.....	.....	.....

- In the query of figure, the valid routing options for different types of tuples (shown on the data flow edges) are as follows:
  - A and C tuples can only be routed to the $A \bowtie B$ and $B \bowtie C$ operator respectively.
  - B tuples can be routed to either of the two join operators.
  - Intermediate AB and BC tuples can only be routed to $B \bowtie C$ and $A \bowtie B$ operator respectively.
  - ABC result tuples are routed to the output.

Tuple routing schemes

An Eddy has a tuple routing scheme that monitors the behaviour of the operators (cost and selectivity) and accordingly routes tuples through the operators.

- *back-pressure*: the processing of a tuple is more slow in a high cost operator than in that of low cost. This generates larger input queue sizes for high costs operators. If the length of input queues is fixed, the Eddy operator is forced to route tuples to an operator of lower cost before routing to those of higher cost.
- *lottery scheduling*: each time a tuple is routed to an operator, it obtains a ticket. When the operator returns a tuple to the Eddy, one ticket is debited. Thus, the number of tickets is used to roughly estimate the selectivity of an operator. When two operators are eligible to process a tuple, the operator with more tickets has higher probability to process it. By doing this, the Eddy is very adaptive and the join order can be changed at runtime.

The Eddy operator

Drawback

(1) At the beginning of the query processing the data source of A is stalled. Thus, $A \bowtie B$ operator is an attractive destination for routing B tuples and the Eddy executes plan $(A \bowtie B) \bowtie C$

(2) Some time later, a great quantity of A tuples arrive and it becomes apparent that the plan $A \bowtie (B \bowtie C)$ is the better choice.

(3) The Eddy switches the routing policy so that subsequent B tuples are routed to $B \bowtie C$ first.

(4) However, the Eddy continues to emulate $(A \bowtie B) \bowtie C$ because of all the previously seen B tuples are still stored in the internal state of the $A \bowtie B$ operator. As A tuples arrive, they must join with these B tuples before the B tuples are joined with C. tuples.

The Eddy continues to emulate an suboptimal plan even after it has switched the routing policy.

The Eddy operator

SteMs

- The State Modules (SteMs) architecture is an extension of the Eddy architecture that ensures that the state stored in the join operators is entirely independent of routing history.
- To this end, SteMs does not store intermediate results.
- The main operator is a SteM, which is instantiated for each attribute of each base relation addressed in the join predicates
- Tuples arriving from each base relation are first built into their own SteM and then used to probe the other relations' SteMs to get the join results.

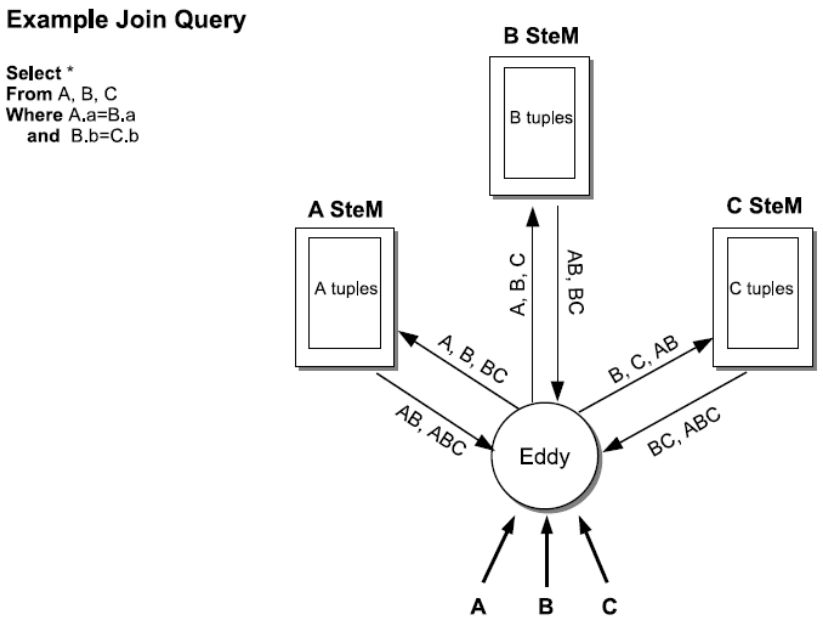
The Eddy operator

Considering the query of the figure, when a new A tuple arrives, it is:

(1) inserted into the A SteM.

(2) probed against B tuples stored in the B SteM to find matching tuples corresponding to $A \bowtie B$.

(3) the resulting AB tuples are probed against the C tuples stored in the C SteM in order to generate ABC results. Intermediate AB tuples are not stored anywhere. Thus, the state accumulated into SteMs is independent of the routing history.



Drawbacks

- **Re-computation of intermediate results**: since intermediate results are not stored anywhere, they are re-computed each time they are needed.
- **Constrained plan choices**: query plans that can be executed for any new tuples are constrained because even if the Eddy knows the existence of an optimal query plan, this plan is not feasible. For example, any new A tuple must join with B tuples (stored in the SteM on B) first and then with C tuples (stored in the SteM on C). This restricts the query plan for new A tuples to be $(a \bowtie B) \bowtie C$. A new optimal query plan such as $a \bowtie (B \bowtie C)$ cannot be proposed because the absence of $B \bowtie C$ tuples.

STAIRs

- The STAIR operator exposes the state stored in the operators and allows to the Eddy manipulate this state in order to reverse any bad routing decisions.
- A STAIR on relation A and attribute a, denote by $A.a$, contains either tuples from A or intermediate tuples that contain a tuples from A.
- To reverse any bad routing decisions made in past, STAIRs perform the following operations:
  - $Demotion(A.a, t, t')$: this operation reduces an intermediate tuple t stored in the STAIR $A.a$ to a sub-tuple t' of that tuple. Intuitively, this operation undoes a tuple that was done earlier during execution.
  - $Promotion(A.a, t, B.b)$: this operation replaces a tuple t with super tuples of that tuple generated using another join in the query. Intuitively, this operation reroutes the tuple t to the new join ordering.

The Eddy operator