PermJoin: An Efficient Algorithm for Producing Early Results in Multi-join Query Plans

Justin J. Levandoski Mohamed E. Khalefa Mohamed F. Mokbel

University of Minnesota Department of Computer Science

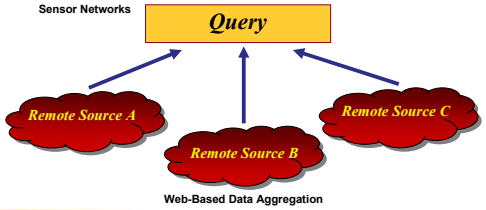
We introduce an efficient algorithm for Producing Early Results in Multi-join query plans (PermJoin, for short). While most previous research focuses only on the case of a single join operator, PermJoin addresses query plans with multiple join operators. PermJoin is optimized to maximize the early overall throughput and to adapt to fluctuations in data arrival rates. PermJoin is a non-blocking operator that is capable of producing join results even if one or more data sources block due to slow or bursty network behavior. Furthermore, PermJoin distinguishes itself from all previous techniques as it: (1) employs a new flushing policy to write in-memory data to disk, once memory allotment is exhausted, in a way that helps increase the probability of producing early result throughput multi-join queries, and (2) employs a novel state manager module that adaptively switches operators between joining in-memory data and disk-resident data in order to maximize overall throughput.

Join Algorithms for Emerging Environments

Scientific Simulation

Sensor Networks

Moving Object Environments

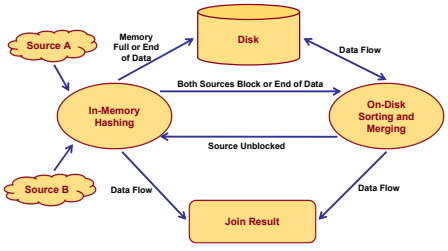


Web-Based Data Aggregation

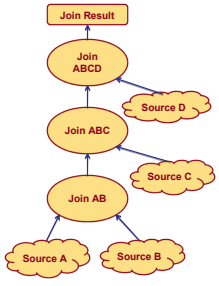
- ❑ Goal: Produce early query feedback in new and emerging environments
- ❑ Constraints
  - ○ Streaming data: not all data available beforehand
  - ○ Sources may block
  - ○ Traditional join algorithms optimized to produce entire result
- ❑ Applications
  - ○ Web-based environment with slow and bursty input with streaming data
  - ○ Sensor networks
  - ○ Scientific simulations taking days to produce large-scale results with need for early results

General Approaches

Single-join query plans: Hash-merge Join



Multi-join query plans: PermJoin



- ❑ Main Idea
  - ○ Collect statistics during query runtime for input sources and data on disk
- ❑ Memory Flushing
  - ○ Consider data at each operator equally, flush data least beneficial to query plan
- ❑ State Manager
  - ○ Place each operator in optimal state to produce high throughput: in-memory, on-disk, or temporary blocking

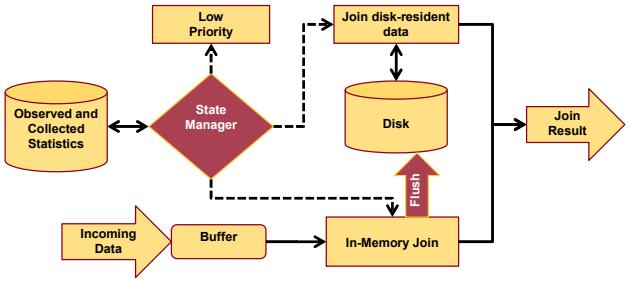
PermJoin Architecture

Join operator can be in *three* states

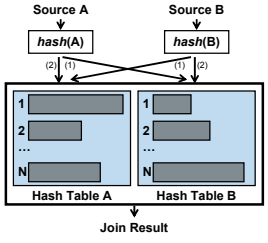
- **In-memory**
  - Joining memory-resident data
- **On-disk**
  - Joining disk-resident data
- **Low priority**
  - Producing results only if resources are available

Consider incoming tuple *Rs*

- **If operator not *in-memory***
  - Rs temporarily buffered
- **If operator in-memory**
  - Rs joined with memory-resident data immediately



In-Memory Join



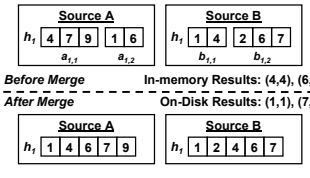
In-Memory: Symmetric-Hash Join

- **Incoming tuple r at Source A**
  - Compute hash(r)
  - Probe table B (produce results)
  - Store in table A
- **Symmetric operator for Source B**

On-Disk: Sort-Merge Join

- **Join different groups**
  - Example: $a_{1,1}$ *and* $b_{1,2}$
- **No need to join similar groups**
  - Example: $a_{1,1}$ *and* $b_{1,1}$

On-Disk Join



Flushing: AdaptiveGlobalFlush

- ❑ Used once hash table memory exhausted
- ❑ Consider all operators in query plan
- ❑ Flush partition groups
  - ○ Symmetric partitions from both sources
- ❑ Based on three characteristics of in-memory data
  - ○ Global contribution: the ability to produce overall results
  - ○ Arrival patterns: changes in data arrival rates at each partition group
  - ○ Data properties: join attribute distribution or whether data is sorted

State Manager

- ❑ Continuous process
- ❑ Traverse query plan to determine optimal state for each operator
- ❑ Fundamentally different than flush algorithm
  - ○ Flushing evicts least-valuable in-memory data
  - ○ Due to changing nature of query, on-disk data may become valuable later in query runtime
    - ○ State manager attempts to find this data to increase early throughput, changing each operator to the appropriate state