Parallel Databases

R&G Chapter 22

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Recap: Joins

- Block-Nested Loop Join
  - Can preemptively eliminate some blocks
- Hash-Join
- Sort/Merge Join
  - Range-partition first
- **Bloom Join**

Bloom Joins

- Based on Bloom Filters
  - A technique for "summarizing" <u>sets</u>.
  - Creates a "sketch" that can be used to speed up <u>set membership</u> tests.
- Summary: Use bloom filters to determine which tuples can participate in an equi-join.

Bloom Filters

		Hash Value							
A	:	0	0	0	0	0	1	0	1
B	:	0	1	0	1	0	1	0	0
C	:	0	0	0	1	1	0	1	0
D	:	0	1	0	0	0	0	0	1

image credit: openclipart.org

Bloom Filters

Hash Value

A	:	0	0	0	0	0	1	0	1
		v	v	v	v	v	v	v	v
B	:	0	1	0	1	0	1	0	0
		v	v	v	v	v	v	v	v
C	:	0	0	0	1	1	0	1	0
		v	v	v	v	v	v	v	v
D	:	0	1	0	0	0	0	0	1
Filter:		0	1	0	1	1	1	1	1

image credit: openclipart.org

Bloom Filters

A	:	0	0	0	0	0	1	0	1
B	:	0	1	0	1	0	1	0	0

C	:	0	0	0	1	1	0	1	0
D	:	0	1	0	0	0	0	0	1

image credit: openclipart.org

Bloom Filters

A :	0	0	0	0	0	1	0	1
B :	0	1	0	1	0	1	0	0
Filter 1:	0	1	0	1	0	1	0	1

C :	0	0	0	1	1	0	1	0
D :	0	1	0	0	0	0	0	1
Filter 2:	0	1	0	1	1	0	1	1

image credit: openclipart.org

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1
Filter 2:	0	1	0	1	1	0	1	1
A :	0	0	0	0	0	1	0	1

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1
Filter 2:	0	1	0	1	1	0	1	1
A :	0	0	0	0	0	1	0	1

hash(A) $\wedge_{\text{bitwise}}$ (Filter X) $\equiv$ hash(A)

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1	
Filter 2:	0	1	0	1	1	0	1	1	
A :	0	0	0	0	0	1	0	1	
	0	0	0	0	0	1	0	1	✓
	0	0	0	0	0	0	0	1	✗

$$\text{hash}(A) \wedge_{\text{bitwise}} (\text{Filter } X) \equiv \text{hash}(A)$$

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1	
Filter 2:	0	1	0	1	1	0	1	1	
A :	0	0	0	0	0	1	0	1	
	0	0	0	0	0	1	0	1	✓
	0	0	0	0	0	0	0	1	✗

$$\text{hash(A)} \wedge_{\text{bitwise}} \text{(Filter X)} \equiv \text{hash(A)}$$

A <u>can not</u> have participated in the construction of F2

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1
Filter 2:	0	1	0	1	1	0	1	1
A :	0	0	0	0	0	1	0	1
B :	0	1	0	1	0	1	0	0
C :	0	0	0	1	1	0	1	0
D :	0	1	0	0	0	0	0	1

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1
Filter 2:	0	1	0	1	1	0	1	1
A :	0	0	0	0	0	1	0	1
B :	0	1	0	1	0	1	0	0
C :	0	0	0	1	1	0	1	0
D :	0	1	0	0	0	0	0	1

Membership Testing

Filter 1:	0	1	0	1	0	1	0	1
Filter 2:	0	1	0	1	1	0	1	1
A :	0	0	0	0	0	1	0	1
B :	0	1	0	1	0	1	0	0
C :	0	0	0	1	1	0	1	0
D :	0	1	0	0	0	0	0	1

D could have participated in the construction of both!

Membership Testing

- No False Negatives:
  - Will never say that a value is not an element of a set when it is.
- May Have False Positives:
  - Might say that a value is in a set when it is not.
  - Can still reduce total number of values tested.

Bloom Join

- Computing Equijoin: R $\bowtie$ S
  - Site 1 stores R, Site 2 stores S
- Site 1 generates a bloom filter for the set of <u>join keys</u> of tuples of R and sends the filter to Site 2.
- Site 2 tests the join keys of all tuples of S against the bloom filter and sends matches to Site 1.
- Site 1 performs a local join.

“Full” Bloom Filters

- A bloom filter that is all 1s is useless.
  - Every element “in the set”; Many false positives.
- Adopt a hash function that generates few ‘1’ bits.
  - E.g., Each bit is the AND of several bits of a hash.
- Use a larger Bloom Filter
  - Rule of thumb: O(N) bits of filter.

Distributed Transactions

Distributed Transactions

- Transactions can update multiple objects
  - … and data is replicated for performance/redundancy
- Isolation challenge: All sites participating in the transaction must be updated.
- Durability challenge: After a failure, some sites may not recover (e.g., Hurricane).

Providing Isolation

- All the strategies discussed for query interleaving work here.
  - Most common: Locking and Versioning.
- Versioning works unchanged.
- Locking presents a new challenge:
  - How do we detect deadlock?

Distributed Deadlock Detection

Waits-For Graphs

Site 1: $$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

Site 2: $$\begin{tikzcd} T1 & T2 \arrow[l] \end{tikzcd}$$

Locally: No Deadlock

Globally: Deadlock

Distributed Deadlock Detection

- Naive Solution: Centralized Locking
    - Single site handles everything
- Slightly Less Naive: Centralized Detection
    - Sites periodically (e.g., every 10s) send local waits-for graphs to central site.
- Expensive, and Non-Scalable

Distributed Deadlock Detection

- Solution #3: Form sites into hierarchies.
  - Lowest tier communicate frequently amongst themselves (e.g., 10s)
  - Next lowest tier communicates less frequently (e.g., 1 min)
  - etc…

Distributed Deadlock Detection

- Solution #4: Timeouts
  - If a transaction waits for longer than some threshold, abort it.
  - Dangerous, can lead to false positives
    - … but NO coordination overheads.

Durability

- New kinds of failure modes:
  - Network Partitions; Some nodes lose connectivity with other nodes.
  - Partial Failures; Some nodes fail permanently, others fail temporarily.
    - Important for replication.
- When have we safely committed?

2-Phase Commit

- Phase 1: Prepare
  - Ensure that all sites can safely commit.
  - Ensure that no site will need to abort.
- Phase 2: Notify
  - Communicate the commit to each site.
- After phase 1 completes successfully, the transaction will never abort.

2-Phase Commit

- One site selected as a coordinator.
  - Initiates the 2-phase commit process.
- Remaining sites are subordinates.

- Only one coordinator per xact.
  - Different xacts may have different coordinators.

2-Phase Commit

- Coordinator sends ‘prepare’ to each subordinate.
- When subordinate receives ‘prepare’, it makes a final decision: Commit or Abort.
  - The transaction is treated as if it committed for conflict detection.
  - The subordinate logs ‘prepare’, or ‘abort’
  - The subordinate responds ‘yes’, or ‘no’

2-Phase Commit

- If coordinator receives 'no' from <u>any</u> subordinate, it tells subordinates to 'abort'.
  - Can treat timeouts as 'no's
- If coordinator receives 'yes' from <u>all</u> subordinates, it tells subordinates to 'commit'
- In both cases, the coordinator first logs the decision and forces the log to local storage.

2-Phase Commit

- Subordinates perform abort or commit as appropriate (logging as in single-site ARIES)
- Subordinates 'ack'nowledge the coordinator.
- The transaction is complete once the coordinator receives all 'acks'.

Recovery

- Network Partition (aka Net-Split)
  - What happens in Phase 1? Phase 2?
- Transient (or Permanent) Failure
  - Coordinator in Phase 1? Phase 2?
  - Subordinate in Phase 1? Phase 2?