Distributed Databases


In part 1 of my lecture, I will use slides extracted from the set of slides provided with the book.

**Goal:** Bring you up to speed on the general area

**Thought exercise:** How does IrisNet fit in this space?

---

**Homogeneous Distributed Databases**

- In a homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests
  - Each site surrenders part of its autonomy in terms of the right to change schemas or software
  - Appears to user as a single system

- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

---

**Horizontal Fragmentation of `account` Relation**

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillside</td>
<td>A-305</td>
<td>500</td>
</tr>
<tr>
<td>Hillside</td>
<td>A-226</td>
<td>336</td>
</tr>
<tr>
<td>Hillside</td>
<td>A-155</td>
<td>62</td>
</tr>
</tbody>
</table>

\[ account := \sigma_{branch-name="Hillside"}(account) \]

<table>
<thead>
<tr>
<th>branch-name</th>
<th>account-number</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valleyview</td>
<td>A-177</td>
<td>205</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-402</td>
<td>10000</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-408</td>
<td>1123</td>
</tr>
<tr>
<td>Valleyview</td>
<td>A-639</td>
<td>750</td>
</tr>
</tbody>
</table>

\[ account := \sigma_{branch-name="Valleyview"}(account) \]
**Vertical Fragmentation of employee-info Relation**

<table>
<thead>
<tr>
<th>branch-name</th>
<th>customer-name</th>
<th>tuple-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillside</td>
<td>Lowman</td>
<td>1</td>
</tr>
<tr>
<td>Hillside</td>
<td>Camp</td>
<td>2</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Camp</td>
<td>3</td>
</tr>
<tr>
<td>Hillside</td>
<td>Kahn</td>
<td>4</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Kahn</td>
<td>5</td>
</tr>
<tr>
<td>Valleyview</td>
<td>Green</td>
<td>6</td>
</tr>
</tbody>
</table>

deposit1 = \( \Pi_{\text{branch-name}, \text{customer-name}, \text{tuple-id}}(\text{employee-info}) \)

<table>
<thead>
<tr>
<th>account number</th>
<th>balance</th>
<th>tuple-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-305</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>A-226</td>
<td>336</td>
<td>2</td>
</tr>
<tr>
<td>A-177</td>
<td>205</td>
<td>3</td>
</tr>
<tr>
<td>A-402</td>
<td>10000</td>
<td>4</td>
</tr>
<tr>
<td>A-155</td>
<td>62</td>
<td>5</td>
</tr>
<tr>
<td>A-408</td>
<td>1123</td>
<td>6</td>
</tr>
<tr>
<td>A-639</td>
<td>750</td>
<td>7</td>
</tr>
</tbody>
</table>

**Advantages of Fragmentation**

- **Horizontal:**
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- **Vertical:**
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.

**Data Replication**

- **Advantages of Replication**
  - Availability: failure of site containing relation \( r \) does not result in unavailability of \( r \) if replicas exist.
  - Parallelism: queries on \( r \) may be processed by several nodes in parallel.
  - Reduced data transfer: relation \( r \) is available locally at each site containing a replica of \( r \).

- **Disadvantages of Replication**
  - Increased cost of updates: each replica of relation \( r \) must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

**Data Transparency**

- **Data transparency:** Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- **Consider transparency issues in relation to:**
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

**Transactions**

- **Transfer $50 from account A to account B**

<table>
<thead>
<tr>
<th>Read(A)</th>
<th>( A := A - 50 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td>( B := B + 50 )</td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

- **ACID properties**
  - **Atomicity:** Either all ops in a transaction are reflected in the DB or none are
  - **Consistency:** Application-specific consistency is preserved for isolated transaction (e.g., \( A+B \) unchanged)
  - **Isolation:** It appears to Ti that Tj executed before it or after it
  - **Durability:** Committed changes persist even on system failures

**Distributed Transactions**

- Locks \( \rightarrow \) concurrency control
### Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

### System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - Network partition
    - A network is said to be partitioned when it has been split into two or more subsystems that lack any connection between them
    - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

### Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - A transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - Not acceptable to have a transaction committed at one site and aborted at another
- The two-phase commit (2PC) protocol is widely used
- The three-phase commit (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol (e.g., sites not blocked waiting for coordinator recovery)

### Two Phase Commit Protocol (2PC)

- Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let $T$ be a transaction initiated at site $S_i$, and let the transaction coordinator at $S_i$ be $C_i$

#### Phase 1: Obtaining a Decision

- Coordinator asks all participants to prepare to commit transaction $T$.
  - $C_i$ adds the records $<\text{prepare } T>$ to the log and forces log to stable storage
  - Sends $\text{prepare } T$ messages to all sites at which $T$ executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - If not, add a record $<\text{not } T>$ to the log and send $\text{abort } T$ message to $C_i$
  - If the transaction can be committed, then:
    - Add the record $<\text{ready } T>$ to the log
    - Force all records for $T$ to stable storage
    - Send $\text{ready } T$ message to $C_i$
Phase 2: Recording the Decision

- T can be committed if Ci received a \textit{ready T} message from all the participating sites; otherwise T must be aborted.
- Coordinator adds a decision record, \textit{<commit T>} or \textit{<abort T>}, to the log and forces record onto stable storage. Once the record is in stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site Si recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contains \textit{<commit T>} record: site executes \textit{redo (T)}
- Log contains \textit{<abort T>} record: site executes \textit{undo (T)}
- Log contains \textit{<ready T>} record: site must consult Ci to determine the fate of T.
  - If T committed, \textit{redo (T)}
  - If T aborted, \textit{undo (T)}
- The log contains no control records concerning T: implies that Si failed before responding to the \textit{prepare T} message from Ci
  - since the failure of Si precludes the sending of such a response Ci must abort T
  - Si must execute \textit{undo (T)}

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
  - The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for T is executing then participating sites must decide on T’s fate:
  1. If an active site contains a \textit{<commit T>} record in its log, then T must be committed.
  2. If an active site contains an \textit{<abort T>} record in its log, then T must be aborted.
  3. If some active participating site does not contain a \textit{<ready T>} record in its log, then the failed coordinator Ci cannot have decided to commit T. Can therefore abort T.
  4. If none of the above cases holds, then all active sites must have a \textit{<ready T>} record in their logs, but no additional control records (such as \textit{<abort T>} or \textit{<commit T>}). In this case active sites must wait for Ci to recover, to find decision.
- Blocking problem: active sites may have to wait for failed coordinator to recover.

Recovery and Concurrency Control

- \textit{In-doubt transactions} have a \textit{<ready T>}, but neither a \textit{<commit T>}, nor an \textit{<abort T>} log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of \textit{<ready T>}, write \textit{<ready T L> L} = list of locks held by T when the log is written (read locks can be omitted).
  - For every in-doubt transaction T, all the locks noted in the \textit{<ready T L>} log record are reacquired.
- After lock reacquisition, transaction processing can resume, the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.
Persistent Messaging

- Motivating example: funds transfer between two banks
  - Two phase commit would have the potential to block updates on the accounts involved in funds transfer
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
    - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue
  - Once transaction sending a message is committed, message must guaranteed to be delivered
  - Guarantee as long as destination site is up and reachable, code to handle undeliverable messages must also be available
    - E.g., credit money back to source account.
  - If sending transaction aborts, message must not be sent

Concurrency Control in Distributed Databases

Timestamping

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a unique timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - Each site generates a unique local timestamp using either a logical counter or the local clock.
  - Global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier.

Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: “disadvantages” transactions
- To fix this problem
  - Define within each site $S_i$ a logical clock ($LC_i$), which generates the unique local timestamp
  - Require that $S_i$ advance its logical clock whenever a request is received from a transaction with timestamp $<x,y>$ and $x$ is greater that the current value of $LC_i$.
  - In this case, site $S_i$ advances its logical clock to the value $x + 1$.

Replication with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency (i.e., without a guarantee of serializability)
- E.g., master-slave replication: updates are performed at a single “master” site, and propagated to “slave” sites.
  - Propagation is not part of the update transaction: its is decoupled
  - May be immediately after transaction commits
  - May be periodic
  - Data may only be read at slave sites, not updated
  - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g., from central office to branch-office
  - Also useful for running read-only queries offline from the main database

Replication with Weak Consistency (Cont.)

- Replicas should see a transaction-consistent snapshot of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
  - E.g., Oracle provides a create snapshot statement to create a snapshot of a relation or a set of relations at a remote site
  - Snapshot refresh either by recomputation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh
Distributed Query Processing

For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.

In a distributed system, other issues must be taken into account:
- The cost of a data transmission over the network.
- The potential gain in performance from having several sites process parts of the query in parallel.

Simple Join Processing

Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented:
- account (depositor, branch)
- account is stored at site $S_1$
- depositor at $S_2$
- branch at $S_3$

For a query issued at site $S_i$, the system needs to produce the result at site $S_i$.

Possible Query Processing Strategies

- Ship copies of all three relations to site $S_i$ and choose a strategy for processing the entire locally at site $S_i$.
- Ship a copy of the account relation to site $S_2$, compute $temp_1 = account \bowtie depositor$ at $S_2$. Ship $temp_1$ from $S_2$ to $S_3$, and compute $temp_2 = temp_1 \bowtie branch$ at $S_3$. Ship the result $temp_2$ to $S_i$.
- Devise similar strategies, exchanging the roles $S_1$, $S_2$, $S_3$.

Must consider following factors:
- amount of data being shipped
- cost of transmitting a data block between sites
- relative processing speed at each site

Distributed Databases

- Homogeneous vs. Heterogeneous, Fragmentation, Replication, Data Transparency
- Distributed Transactions & Two Phase Commit
- Concurrency Control: Timestamping, Weak Consistency
- Distributed Query Processing
- Many other issues...
IrisNet Query Processing Goals (I)

- Data transparency
  - Logical view of the sensors as a single queriable unit
  - Logical view of the distributed DB as a single centralized DB
  - Exception: Query-specified tolerance for stale data
- Flexible data partitioning/fragmentation
- Update scalability
  - Sensor data stored close to sensors
  - Can have many leaf OAs

IrisNet Query Processing Goals (II)

- Low latency queries & Query scalability
  - Direct query routing to LCA of the answer
  - Query-driven caching, supporting partial matches
  - Load shedding
  - No per-service state needed at web servers
- Support query-based consistency
  (Global consistency properties not needed for common case)
- Use off-the-shelf DB components

Still to do:
Replication, Robustness, Other consistency criteria, Self-updating aggregates, Historical queries, Image queries...

XML & XPATH

- Previously, distributed DBs studied mostly for relational databases
- IrisNet Data stored in XML databases:
  - Supports a heterogenous mix of self-describing data
  - Supports on-the-fly additions of new data fields
- IrisNet Queries in XPATH:
  - Standard XML language with good DB support
  (Prototype supports the unordered projection of XPATH 1.0)
Outline
- IrisNet query processing overview
- QEG details
- Data partitioning & caching details
- Extensions
- Related work & conclusions

QEG Challenges
- OA’s local DB can contain any subset of the nodes (a fragment of the overall service DB)
- Quickly determining which part of an (XPATH) query answer can be answered from an XML fragment is a challenging task, not previously studied
  - E.g., can this predicate be correctly evaluated?
  - Is the result returned from the local DB complete?
  - Where can the missing parts be gathered?
- Traditional approach of maintaining and using “view” queries is intractable

QEG Solutions
- Instead of using view queries, tag the data itself
  - IrisNet tags the nodes in its fragment with status info, indicating various degrees of completeness
- Maintains partitioning/tagging invariants
  - E.g., when gather data, generalize subquery to fetch partitionable units
  - Ensure that fragment is a valid XML document
- For each missing part, construct global name from its id chain & do DNS lookup
  - Specialize subquery to avoid duplications & ensure progress

QEG Solutions (cont)
- XPATH query converted to an XSLT program that walks the local XML document & handles the various tags appropriately
  - Conversion done without accessing the DB
  - Returning subqueries are spliced into the answer document
Nesting Depth

Query for the cheapest parking spot in block 1 of Oakland:

```
/usRegion[@id='NE']/state[@id='PA']/county[@id='Allegheny'/
city[@id='Pittsburgh']/neighborhood[@id='Oakland']
/block[@id='1']/parkingSpace[not (price > ../parkingSpace/price)]
```

Nesting depth = 1

- If the individual parkingSpaces are owned by different sites (and no useful caching), no one site can evaluate the predicate
  - Currently, block 1 fetches all its parkingSpaces

Outline

- IrisNet query processing overview
- QEG details
- Data partitioning & caching details
- Extensions
- Related work & conclusions

Data Partitioning

- IrisNet permits a very flexible partitioning scheme for distributing fragments of the (overall) service database among the OAs
- "id" attribute defines split points ("IDable nodes")
  - Minimum granularity for a partitionable unit
  - ID of a split node must be unique among its siblings
  - Parent of a non-root split node must also be a split node
- An OA can own (and/or cache) any subset of the nodes in the hierarchy, as long as
  - Ownership transitions occur at split points
  - All nodes owned by exactly one OA

Data Partitioning

- Data fragment at an OA stored as a single XML document in the OA’s local XML database
- The ids on the path from the root to a split node form a globally unique name
- Global name to OA mapping:
  - Store in DNS the IP address of the OA
    - pittsburgh.allegheny.pa.ne.parking.intel-iris.net -> 128.2.44.67
  - When change ownership, just need to update DNS
  - Initially, overall service database on a single OA
    - Command line partitioning, or specify partitioning order

Local Information

- **Local ID information** of an IDable node N
  - ID of N
  - IDs of all its IDable children
- **Local information** of an IDable node N
  - All attributes of N
  - All its non-IDable children & their descendants
  - The IDs of its IDable children
  ```
  <block @id='1' @status='ownsthis'>
    <address>400 Craig</address>
    <parkingSpace @id='1' @status='complete'>
      <available>false</available>
    </parkingSpace>
    <parkingSpace @id='2' @status='IDcomplete'>
      <available>true</available>
    </parkingSpace>
  </block>
  ```

Local Information & Status

- **Local ID information, Local information**
- **ID**: Each site must store the local info for the nodes it owns
  ```
  <block @id='1' @status='ownsthis'>
    <address>400 Craig</address>
    <parkingSpace @id='1' @status='complete'>
      <available>true</available>
    </parkingSpace>
    <parkingSpace @id='2' @status='IDcomplete'>
      <available>true</available>
    </parkingSpace>
  </block>
  ```
  ```
  ```
```
What Invariants & Tags Accomplish

• If a site has information about a node (beyond just its ID), it knows at least
  • the IDs of all its IDable children
  • the IDs of all its ancestors & their siblings
• Each node can answer query or it can construct the global name of the parts that are missing

Caching

• A site can add to its document any fragment such that
  • C1: The document fragment is a union of local info or local ID info for a set of nodes
  • C2: If the fragment contains local info or local ID info for a node, it also contains the local ID info for its parent
    • This maintains I1 and I2
• IrisNet generalizes subqueries to fetch the smallest superset of the answer that satisfies C1 & C2
  • Thus, all subquery results can safely be cached

Outline

• IrisNet query processing overview
• QEG details
• Data partitioning & caching details
• Extensions
• Related work & conclusions

Cache Consistency

• All data is time stamped
  • Include timestamp field in XML schema
  • When cache data, also cache its time stamp
• Queries specify a freshness requirement
  • "I want data that is within the last minute"
  • "Have you seen Joe?" - today? this morning? last 10 minutes?
• QEG procedure ignores too-stale data
  • Carefully designed set of cache invariants & tags ensure that the correct answer is returned
  • Exploring other consistency conditions

Other Extensions

• Ownership changes (e.g., on-the-fly load balancing)
• Schema changes
• Speeding up XSLT processing
• Smarter processing of non-zero nesting depths (to do)
• Other consistency criteria (to do)
• Load balancing & cache eviction policies (to do)

Outline

• IrisNet query processing overview
• QEG details
• Data partitioning & caching details
• Extensions
• Related work & conclusions
Synopsis of Related Work

- Ad Hoc Local Networks of Smart Dust
  - E.g., Motes (IR Berkeley), Estrin et al, Labscape (IR Seattle)

- XML-based Publish-Subscribe in Networks
  - E.g., Snoeren et al, Franklin et al

- Distributed / Federated Databases
  - E.g., Breitbart et al, Adali et al, distributed transactions

- Querying in Networks of Numerical Sensors
  - E.g., Cougar, Fjords, Mining time series, TAG

Related Work - More Details

- TinyDB, Tiny aggregation [Madden et al '02]

- Cougar [Bonnet et al '01] - time series sensor DB

- Fjords [Madden & Franklin '02] - sensor proxy DB

- PIER [Harren et al '01] - queries over DHTs

- Piazza [Gribble et al '01] - data placement in P2P

Conclusions: IrisNet QP

- Data transparency - distributed DB hidden from user

- Flexible data partitioning/fragmentation

- Update scalability
  - Sensor data stored close to sensors: Can have many leaf OAs

- Low latency queries & Query scalability
  - Direct query routing to LCA of the answer
  - Query-driven caching, supporting partial matches
  - Load shedding: No per-service state needed at web servers

- Support query-based consistency

- Use off-the-shelf DB components