Misc: Distributed Transactions; Object-oriented and Object-relational Databases

Amol Deshpande
CMSC424

Spring 2020 – Online Instruction Plan

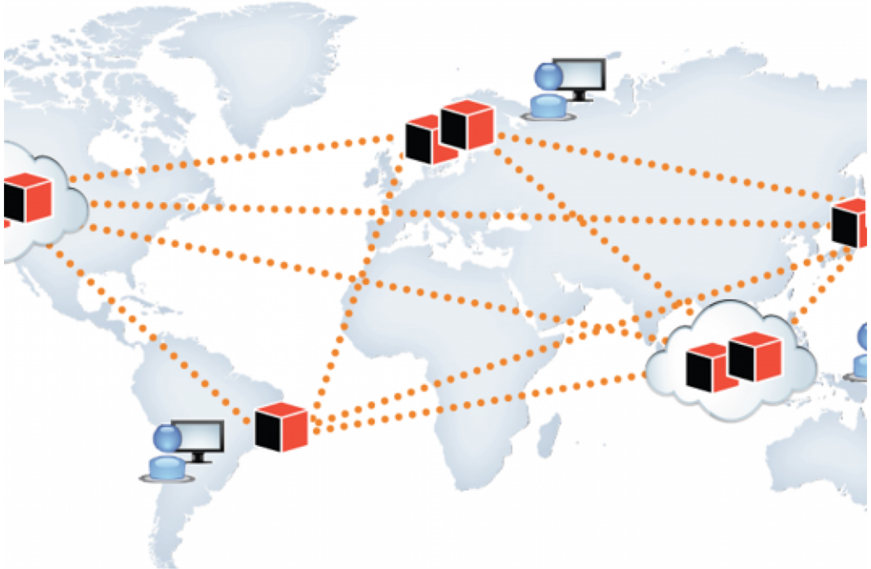
- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
- Week 5: Transactions 2
- Week 6: Homework Due May 8
  - Transactions: Recovery
  - Misc 1: Distributed Transactions, and Object-oriented/Object-relational databases
  - Misc 2: OLAP and Data Cubes

Distributed Transactions

- Book Chapters
  - 19.1-19.4, 19.6: at a fairly high level
- Key topics:
  - Distributed databases and replication
  - Transaction processing in distributed databases
  - 2-Phase Commit
  - Brief discussion of other protocols including Paxos

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
  - Or not – lot of variations here
- Transactions may access data at one or more sites
  - Because of replication, even updating a single data item involves a "distributed transaction" (to keep all replicas up to date)



Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites

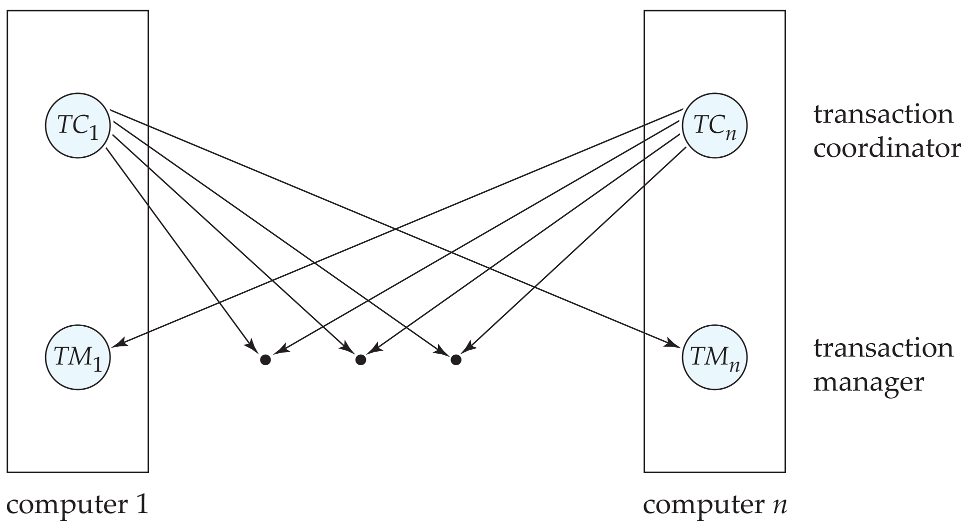
- Advantages:
  - **Availability**: failures can be handled through replicas
  - **Parallelism**: queries can be run on any replica
  - **Reduced data transfer**: queries can go to the "closest" replica

- Disadvantages:
  - **Increased cost of updates:** both computation as well as latency
  - **Increased complexity of concurrency control:** need to update all copies of a data item/tuple

- **Typically we use the term "data items", which may be tuples or relations or relation partitions**

Distributed Transactions

- Transaction may access data at several sites
  - As noted, single data item update is also a distributed transaction
- **Each site has a local transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the concurrent execution of the transactions
- **Each site has a transaction coordinator,** which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing sub-transactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site -- transaction may commit at all sites or abort at all sites.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - 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
- **Two-phase *commit* (2PC)** protocol is widely used
- **Three-phase commit (3PC)** protocol
  - Handles some situations that 2PC doesn't
  - Not widely used
- **Paxos**
  - Robust alternative to 2PC that handles more situations as well
  - Was considered too expensive at one point, but widely used today
- **RAFT**: Alternative to Paxos

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

Two Phase Commit Protocol (2PC)

Coordinator Log	Messages	Subordinate Log
	PREPARE $\rightarrow$	
		prepare*/abort*
	$\leftarrow$ VOTE YES/NO	
commit*/abort*		
	COMMIT/ABORT$\rightarrow$	
		commit*/abort*
	$\leftarrow$ ACK	
end		

Goal: Make sure all ”sites” commit or abort

Assumption: Some log records can be “forced” (denote * above)

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **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 <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record 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 S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: txn had completed, nothing to be done
- Log contains <**abort** *T*> record: txn had completed, nothing to be done
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*); write <**commit** *T*> record
  - If *T* aborted, **undo** (*T*)
- The log contains no log records concerning *T*:
  - Implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response, coordinator C_1 must abort *T*
  - S_k must execute **undo** (*T*)

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 <**commit** T> record in its log, then T must be committed.
  2. If an active site contains an <**abort** T> record in its log, then T must be aborted.
  3. If some active participating site does not contain a <**ready** T> record in its log, then the failed coordinator C_i cannot have decided to commit T.
     - Can therefore abort T; however, such a site must reject any subsequent <**prepare** T> message from C_i
  4. If none of the above cases holds, then all active sites must have a <**ready** T> record in their logs, but no additional control records (such as <**abort** T> of <**commit** T>).
     - In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

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

More…

- Three-phase Commit
  - 2PC can't handle failure of a coordinator well – everything halts waiting for the coordinator to come back up
  - Three-phase commit handles that through another phase
- Paxos and RAFT
  - Solutions for the "consensus problem": get a collection of distributed entities to "choose" a single value
    - In case of transaction, you are choosing abort/commit
  - Fairly complex, but well-understood today
  - Widely used in most distributed systems today
  - See the Wikipedia pages
  - A nice recent paper: **Paxos vs Raft: Have we reached consensus on distributed consensus? – Heidi Howard, 2020**

Object-oriented and Object-relational

- Book Chapters
  - Chapter 22: at a fairly high level
- Key topics:
  - Why Objects?
  - Object-oriented
  - Object-relational

Motivation

- Relational model:
  - Clean and simple
  - Great for much enterprise data
  - But lot of applications where not *sufficiently rich*
    - Multimedia, CAD, for storing *<u>set data</u>* etc
- Object-oriented models in programming languages
  - Complicated, but very useful
    - Smalltalk, C++, now Java
  - Allow
    - Complex data types
    - Inheritance
    - Encapsulation
- People wanted to manage objects in databases.

History

- In the 1980's and 90's, DB researchers recognized benefits of objects.
- Two research thrusts:
  - OODBMS: extend C++ with transactionally persistent objects
    - Used to be a niche Market
    - CAD etc.
    - More recently, made a comeback as a JSON, Graph Databases
      - But those usually have a query language and look more like ORDBMS
  - ORDBMS: extend Relational DBs with object features
    - Much more common
    - Efficiency + Extensibility
    - SQL:99 support
- Postgres – First ORDBMS
  - Berkeley research project
  - Became Illustra, became Informix, bought by IBM

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
   (firstname    varchar(20),
    lastname     varchar(20))
    final

create type Address as
   (street      varchar(20),
    city        varchar(20),
    zipcode     varchar(20))

    not final
```

  - Note: **final** and **not final** indicate whether subtypes can be created
- Structured types can be used to create tables with composite attributes

```
create table person (
    name        Name,
    address     Address,
    dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- **User-defined row types**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
```

- Can then create a table whose rows are a user-defined type

```
create table customer of CustomerType
```

- Alternative using **unnamed row types**.

```
create table person_r(
    name     row(firstname  varchar(20),
                 lastname   varchar(20)),
    address  row(street     varchar(20),
                 city       varchar(20),
                 zipcode    varchar(20)),
    dateOfBirth date)
```

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

          returns interval year

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

          returns interval year

          for *CustomerType*

  begin

          return *onDate* - **self**.*dateOfBirth*;

  end

- We can now find the age of each customer:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department   varchar(20))
```

-

```
create type Teacher
  under Person
  (salary       integer,
   department   varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Array and Multiset Types in SQL

- Example of array and multiset declaration:

create type *Publisher* **as**
  (*name* **varchar**(20),
  branch **varchar**(20));

create type *Book* **as**
  (*title* **varchar**(20),
  author_array **varchar**(20) **array** [10],
  pub_date **date**,
  publisher *Publisher*,
  keyword-set **varchar**(20) **multiset**);

create table *books* **of** *Book;*

Creation of Collection Values

- Array construction

  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets

  multiset [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the books relation:

  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

- To insert the preceding tuple into the relation books

  insert into *books*
  values
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’]);

Querying Collection-Valued Attributes

- To find all books that have the word "database" as a keyword,

  select *title*
  from *books*
  where 'database' **in** (**unnest**(*keyword-set*))

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

  select *author_array*[1], *author_array*[2], *author_array*[3]
  from *books*
  where *title* = `Database System Concepts'

- To get a relation containing pairs of the form "title, author_name" for each book and each author of the book

  select *B.title, A.author*
  from *books* **as** *B*, **unnest** (*B.author_array*) **as** *A* (*author*)

- To retain ordering information we add a **with ordinality** clause

  select *B.title, A.author, A.position*
  from *books* **as** *B*, **unnest** (*B.author_array*) **with ordinality as** *A* (*author, position*)

Path Expressions

- Find the names and addresses of the heads of all departments:

  select *head* –>*name*, *head* –>*address*
  from *departments*

- An expression such as "head–>name" is called a **path expression**
- Path expressions help avoid explicit joins
  - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
  - Makes expressing the query much easier for the user

An Alternative: OODBMS

- Persistent OO programming
  - Imagine declaring a Java object to be "persistent"
  - Everything reachable from that object will also be persistent
  - You then write plain old Java code, and all changes to the persistent objects are stored in a database
  - When you run the program again, those persistent objects have the same values they used to have!
- Solves the "impedance mismatch" between programming languages and query languages
  - E.g. converting between Java and SQL types, handling rowsets, etc.
  - But this programming style doesn't support declarative queries
    - For this reason (??), OODBMSs haven't proven popular
- OQL: A declarative language for OODBMSs
  - Was only implemented by one vendor in France (Altair)

OODBMS

- Currently a Niche Market
  - Engineering, spatial databases, physics etc…
- Main issues:
  - Navigational access
    - Programs specify go to this object, follow this pointer
  - Not declarative
- Though advantageous when you know exactly what you want, not a good idea in general
  - Kinda similar argument as *network databases vs relational databases*

Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, high performance.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system
- Note: Many real systems blur these boundaries
  - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

Summary, cont.

- ORDBMS offers many new features
  - but not clear how to use them!
  - schema design techniques not well understood
    - No good logical design theory for non-1st-normal-form!
  - query processing techniques still in research phase
    - a moving target for OR DBA’s!

- OODBMS
  - Has its advantages
  - Niche market
  - Lot of similarities to XML as well…