Database Systems
CSE 344

Lecture 24: ORM & Final Review

Announcements

- Final on Friday in Class!
- Please complete course evaluations!
  - Summer schedule (what did you think)
  - Feedback for me
  - Deadline Thursday Aug 17

Object Relational Mapper

- Application code must mirror database relations.
- User.java
- Flight.java

```
/** Stores information about a user in the database. */
public class User {

  /** Stores the ID of the authenticated user. */
  public final int id;

  /** Stores the handle of the authenticated user. */
  public final String handle;

  /** Stores the full name of the authenticated user. */
  public final String fullName;

  /** Creates a User with the given properties. */
  public User(int id, String handle, String fullName) {
    this.id = id;
    this.handle = handle;
    this.fullName = fullName;
  }
}
```

Object Relational Mapper

- Application code must mirror database relations.
- Exp: User.java, Flight.java
- This creates repetition in code and more work for developers.
- Solution: Write helper code to translate objects into database tables and back.

Key Idea: Object Relational Mapper: ORM

Object Relational Mapper

- Language specific implementations
  - Java: Hibernate, ActiveJDBC
  - Python: SQLAlchemy, Django.db
  - Ruby on Rails

Django Example

```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Django Example

- QuerySet Managers
  - .create: create a new object
  - .get: fetch a single object
    - keywords go into the where clause
  - filter: fetch multiple objects based on arguments
    - .filter(pub_date__lt=dt.now()) -- all questions published before today.
  - RelatedManagers for foreign keys
  - Database migrations
  - Same code works with sqlite, MySQL and Postgress

ORM: Pros and Cons

- Pros
  - Makes rapid development easy
  - Handles the basic CRUD (create, read, update, delete).
  - Reduces code duplication and work needed to keep database code inline with application code
- Cons
  - Obfuscates SQL - if you don’t know SQL can lead to bad design
  - More complex queries will always require SQL

Final Exam

- Friday, August 18 2:20 - 3:20
- This room
- Closed books, no phones, no computers
- Allowed 2 pages of notes (both sides, 8+pt font)
  – but focus of the test will not be memorization
- Primary focus on the second half
  – More like a “second midterm”

Course Topics

1. Relational Data
2. DB Applications: Design & Implementation
3. Semistructured Data
4. DBMS Implementation
5. Big Data Systems

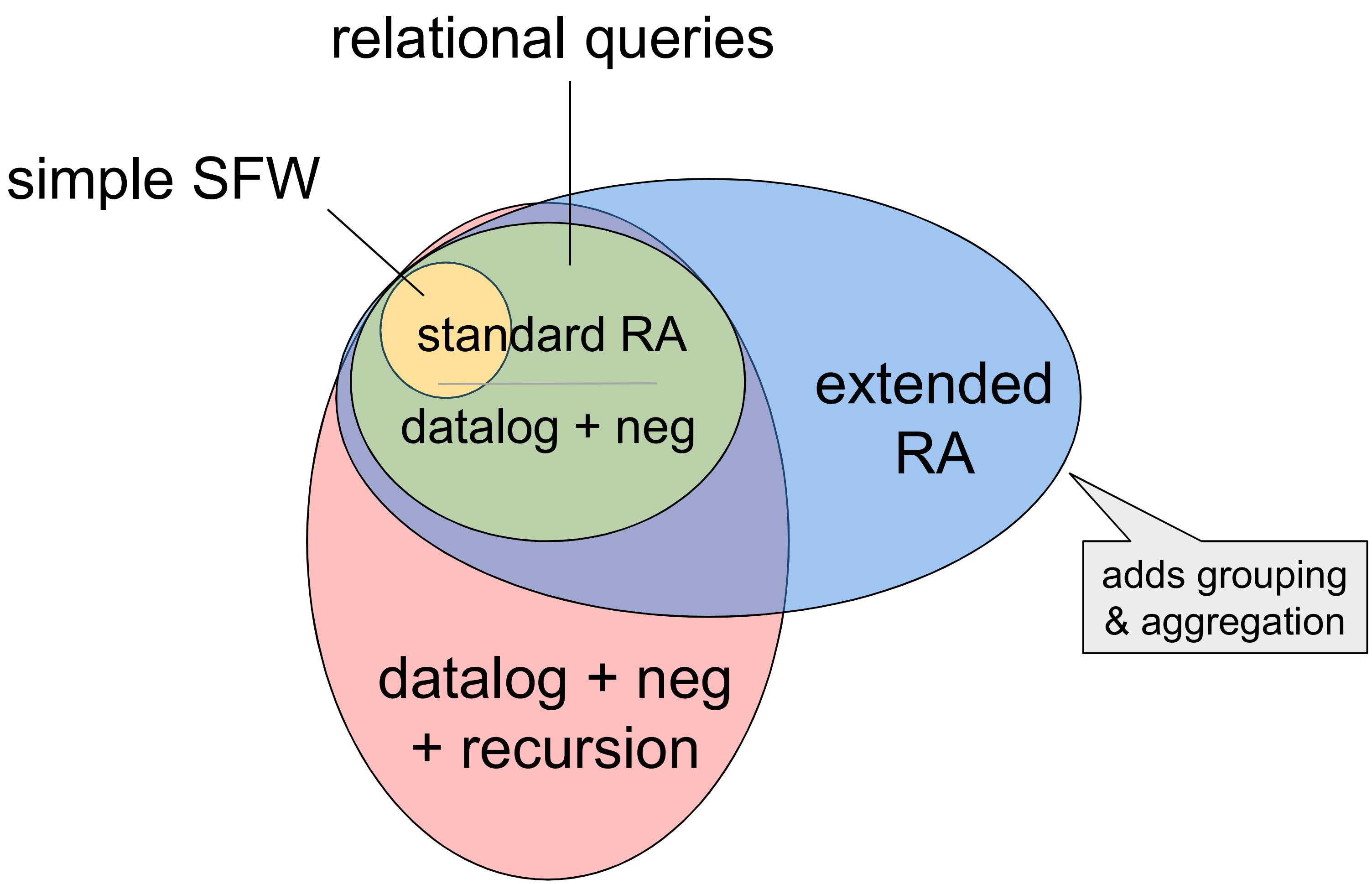
Relational Data

1a. Relational Data Model

- tables with schemas
  - types for attributes
  - primary, secondary, and foreign keys
  - other constraints
- set semantics
  - each tuple is either in the table or not

1b. Relational Queries

- relational query = expressible in standard RA
  - RA = datalog+neg, also expressible with SQL
- simple SELECT-FROM-WHERE is a subset
  - includes joins but not subqueries
  - always monotone while RA isn’t (e.g. set difference)
- extended RA adds grouping & aggregation
  - (also uses bag semantics)
- datalog adds recursion


relational queries
simple SFW
standard RA
datalog + neg
extended RA
adds grouping & aggregation
datalog + neg + recursion

1c. Datalog (not on Final)

- data comes from **facts** and **rules**
  - $P(a_1, \ldots, a_n)$.
  - $Q(a_1, \ldots, a_n)$:- $R1(a_i, b_k, \ldots)$, $R2(a_j, b_l, \ldots)$, ...
- head is a fact iff there is *some* way to set b_k's so that all terms in the body are facts
  - variables only appearing in body (b_k's) are *existential*
- can be translated to SQL
  - must be possible since datalog equivalent to RA
  - but we didn't discuss the details…

DB Applications:
Design & Implementation

2a. DB Design Process

Conceptual Model:

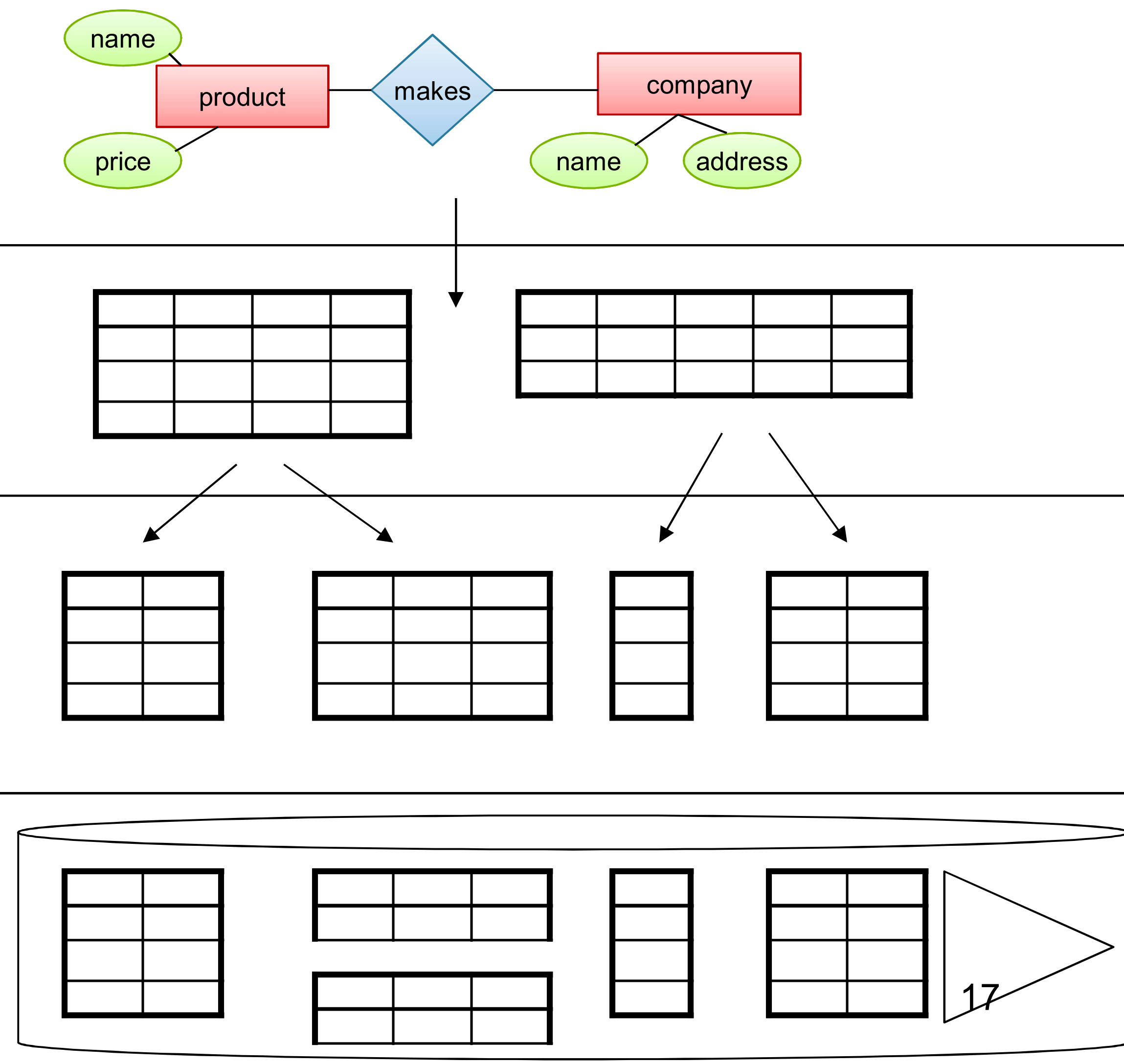
Relational Model:
Tables + constraints
And also functional dep.

Normalization:
Eliminates anomalies

Conceptual Schema

Physical storage details

Physical Schema



2a. DB Design Process

- E/R Diagrams
  - entity sets, relations, & subclasses
  - map each to relations
    - multiple ways to do this (many-many, one-many) only need to know the approach from class
  - design principles:
    - model accurately
    - neither too few nor too many entities

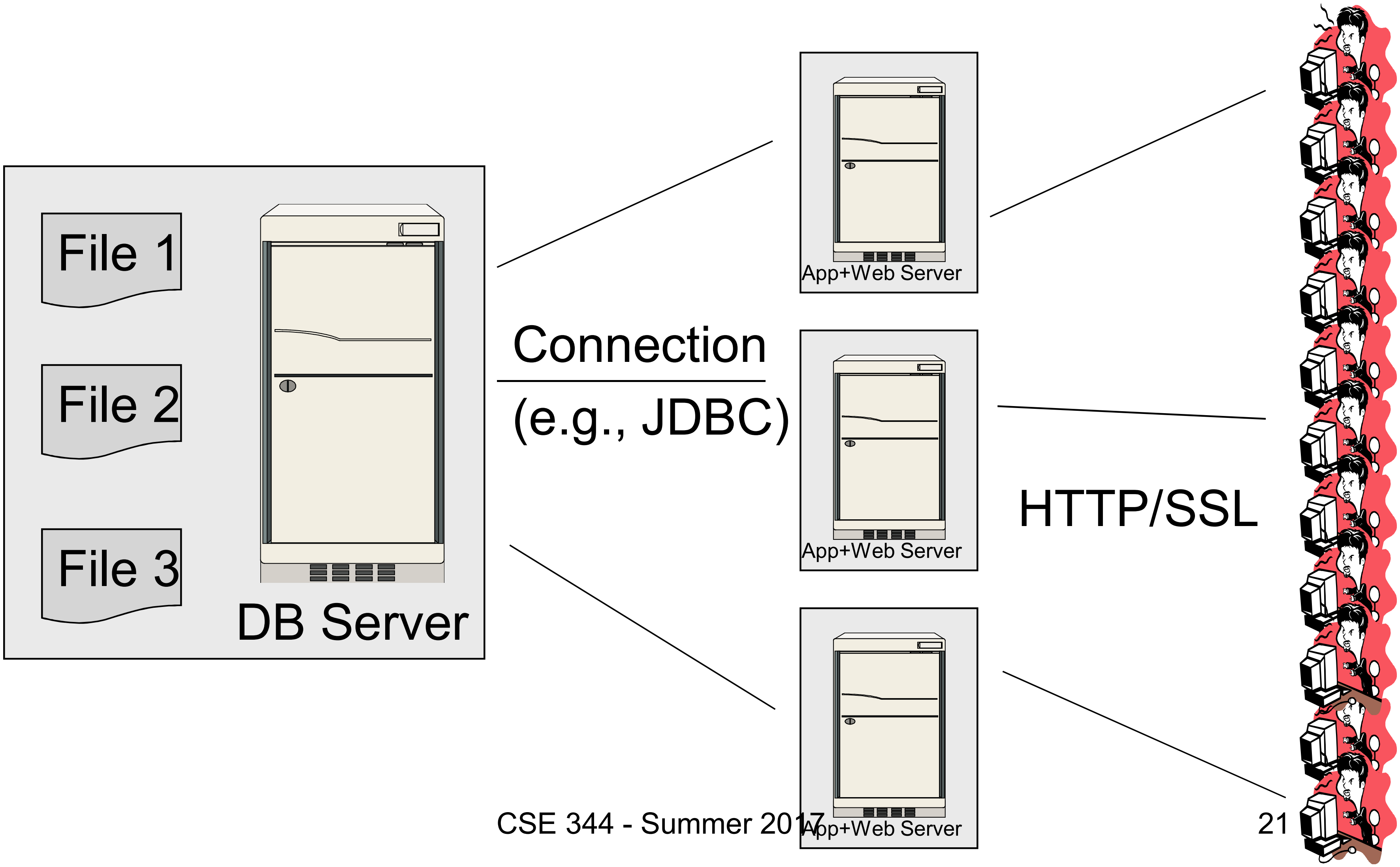
2a. DB Design Process

- Constraints
  - key, single-value, referential & other constraints
    - other includes, e.g., positivity and non-null constraints
- Normalization
  - eliminates anomalies
    - redundancy, update, and deletion anomalies
  - are indicated by “bad” functional dependencies
  - apply BCNF decomposition to remove them
    - these decompositions are never lossy (others can be)

2b. DB Application Implementation

- JDBC
  - connect to DB from Java
  - send SQL statements
  - use transactions
- 3-tiered architecture for web applications

3-Tiered Architecture



2b. DB Application Implementation

- JDBC
  - connect to DB from Java
  - send SQL statements
  - use transactions
- 3-tiered architecture for web applications
  - usually JSON data btw web server & browser/phone
  - why not use JSON to the DB too?
    - otherwise, we need to translate JSON to relational

Semistructured Data

3a. Semistructured Data Model

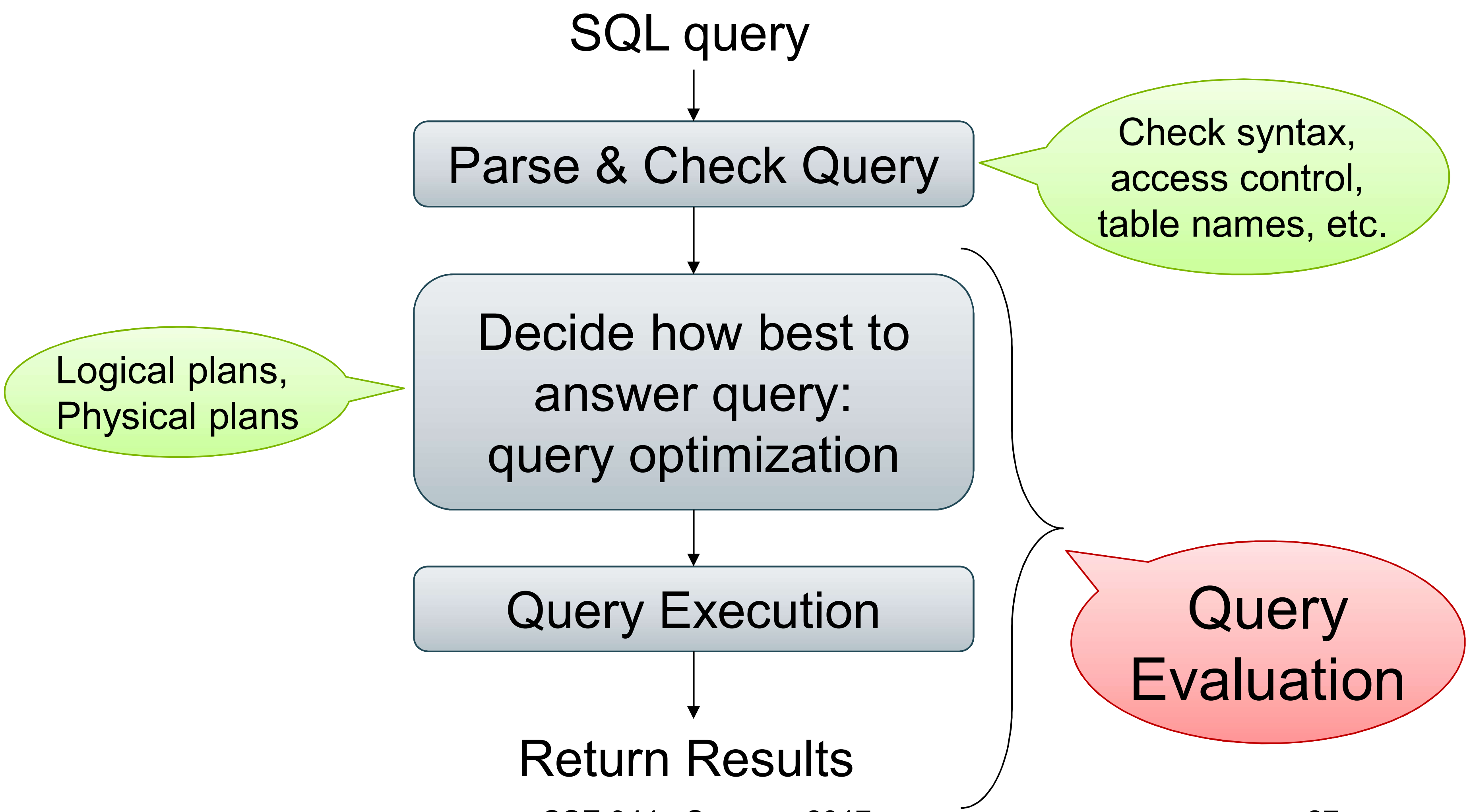
- tree structured data: JSON, XML, etc.
- data is self-describing
  - so schema is not necessary
- easy to map relation to JSON but not opposite

DBMS Implementation

4a. Storage & Indexing (not on Final)

- B+ tree & hash indexes
  - B+ tree index allows searching by key prefixes also
- understand when an index can be used
  - (separate question from whether it improves perf)
- clustered vs unclustered
  - clustered always speeds up query
    but only one index per table can be clustered
  - unclustered only speeds up if <1% tuples match

Query Evaluation Steps



4b. Query Optimization (not on Final)

- main cost is disk access
- many logical plans, many physical plans
  - logical plans are RA expressions with desired result
  - physical plans include e.g. choice of join algorithm
    - hash, sorted merge, and (block refined) nested loop joins
- cost of many operations depends on selectivity
- optimization problem is hard
  - saw SQL Server do poorly in homework problems
- realistic goal is to avoid really bad plans

4c. Transactions

- goal to allow many clients to run simultaneously
  - OLTP workload: lots of clients with small read/writes
- need to provide ACID properties
  - atomic: execute all SQL statements or none
  - consistent: finish with all constraints satisfied
  - isolation: behavior same as if one-at-a-time use
  - durable: committed result are permanent (‘til changed)
- consistency maintained by checking constraints
- durability maintained by writing to disk(s)

4c. Transactions II

- isolation achieved through serializable schedules
  – serializable means same behavior as a serial schedule
  – conflict serializable means non-conflicting read/writes can be swapped to make schedule serial
    - stronger than (so implies) serializable
- locks ensure conflict serializability if 2PL used
  – multiple read locks, only one write lock
    - becomes 4 types in SQLite (a good design)
  – lock granularity from (parts of) rows to tables to DB
  – …

4c. Transactions III

  - strict 2PL: no unlocks before commit/rollback
    - needed for isolation if txns can roll back
  - can produce deadlocks (as seen in homework)
  - need more to prevent phantom rows
    - phantom is a new row that shows up in a table
    - predicate locks are one solution (but expensive)

- multi-version concurrency control is alternative
- default isolation level is usually not serializable
  - faster perf but harder to write app (i.e., bugs likely)

Isolation Levels in SQL

- READ UNCOMMITTED ("Dirty reads")
  - Write: strict 2PL, Read: none
- READ COMMITTED ("Committed reads")
  - Write: strict 2PL, Read: Short-term
- REPEATABLE READ ("Repeatable reads")
  - Write: strict 2PL, Read: strict 2PL
- SERIALIZABLE (Serializable transactions)
  - No phantom reads

ATOMIC

Systems for Big Data

5a. NoSQL Systems

- goal to support heavy OLTP workloads
- provides simplified data model
  - key-value pairs, documents, or extensible records
- limited support for transactions
  - usually pair/document/record level
  - (some support for record groups… all on one node)
- partition data across nodes for scale
- replicate data to survive node failures

5b. Parallel Processing Systems

- for OLAP workloads (big reads, no txns)
- MapReduce
  - programming model is one-to-many *map* function, shuffle sort (grouping), one-to-many *reduce* function
  - no built-in RA operators
    - but easy to implement since since shuffle sort is provided
  - stores intermediate data on disk
    - reasonable if input/output is also to disk (otherwise too slow)
  - deals with stragglers by running backup map tasks

5b. Parallel Processing Systems II

- Spark/Scala
  - executes a dataflow pipeline using many nodes
  - Spark handles failure by recomputing not replicating
- Spark SQL
  - map SQL ~> extended RA ~> dataflow pipeline
  - same approach can be used on any dataflow engine

5b. Parallel Processing Systems III

- existing systems do not optimize well
  - none do real cost-based optimization
  - Spark only performs small, syntactic optimizations
    - one exception: choice of parallel vs broadcast join
  - Spark has no indexes
  - all require manual tuning

5c. Relational Parallel Databases

- support both OLTP and OLAP
- goal: more nodes => faster or allow more data
  - speed up or scale up
- different architectures
  - shared memory (SQL Server etc.): limited scale
  - shared disk (mostly Oracle): limited scale
  - shared nothing: really scales (so our focus)
    - won out in academic research (started in 1980s)
    - basis for parallel processing systems (see previous slides)

5c. Parallel Databases II

- Partition data across nodes (hash, range, etc.)
- Query evaluation
  - only one new element: reshuffle
    - move tuples to nodes based on values in certain columns
    - basically same as shuffle sort of MapReduce
    - use to implement all extended RA operations
  - linear speed up or scale up in principle
  - in practice, stragglers are a problem (though see MapReduce)
  - new problem: skewed data
    - may not all fit in memory of one node

SQL (Everywhere)

5. SQL

- CREATE TABLE …
  - PRIMARY KEY, UNIQUE, FOREIGN KEY
  - CHECK (constraints) on columns or tuples
- CREATE [CLUSTERED] INDEX … ON ...
- INSERT INTO …
- UPDATE … SET ... WHERE ...
- DELETE FROM ... WHERE …

5. SQL (cont.)

- SELECT …
  - JOINs: inner vs outer, natural
  - GROUP BY, sum, count, avg, etc.
  - ORDER BY
- SET ISOLATION LEVEL …
- BEGIN TRANSACTION
- COMMIT / ROLLBACK