W4111 – Introduction to Databases
Section 002, Spring 2024
Lecture 11
Module II (4), NoSQL (4)

W4111 – Introduction to Databases
Section 002, Spring 2024
Lecture 11
Module II (4), NoSQL (4)

Say something if you are in OHs and I an not paying attention.

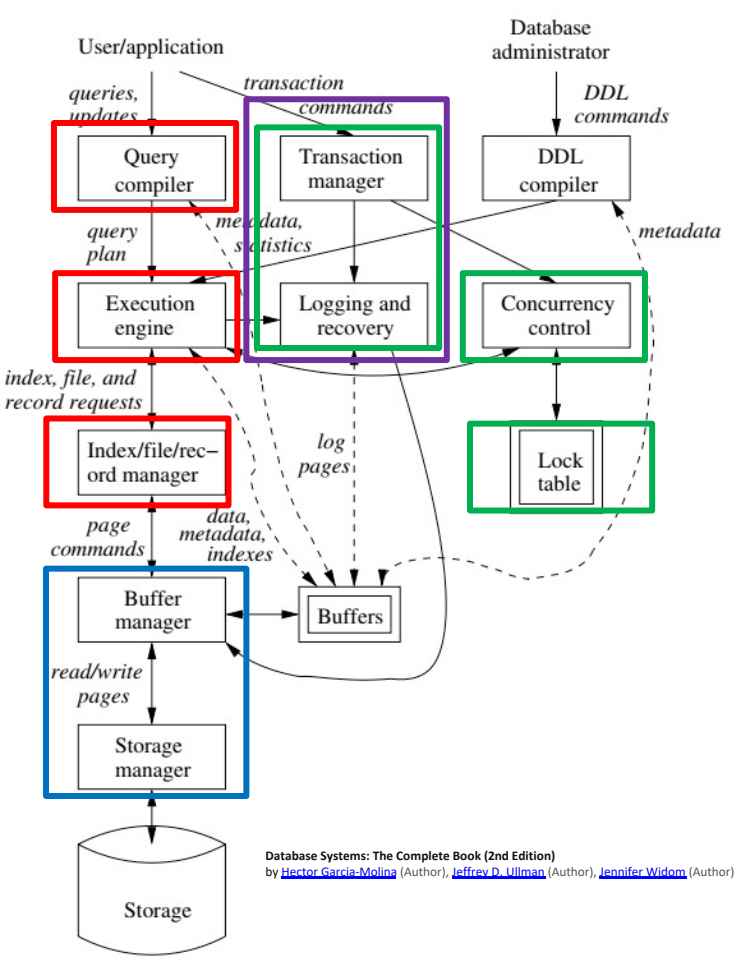
We will start in a couple of minutes.

Module II, Part 4

Reminder

Transactions

- Find things quickly.
- Load/Save quickly.
- Transactions
- Durability

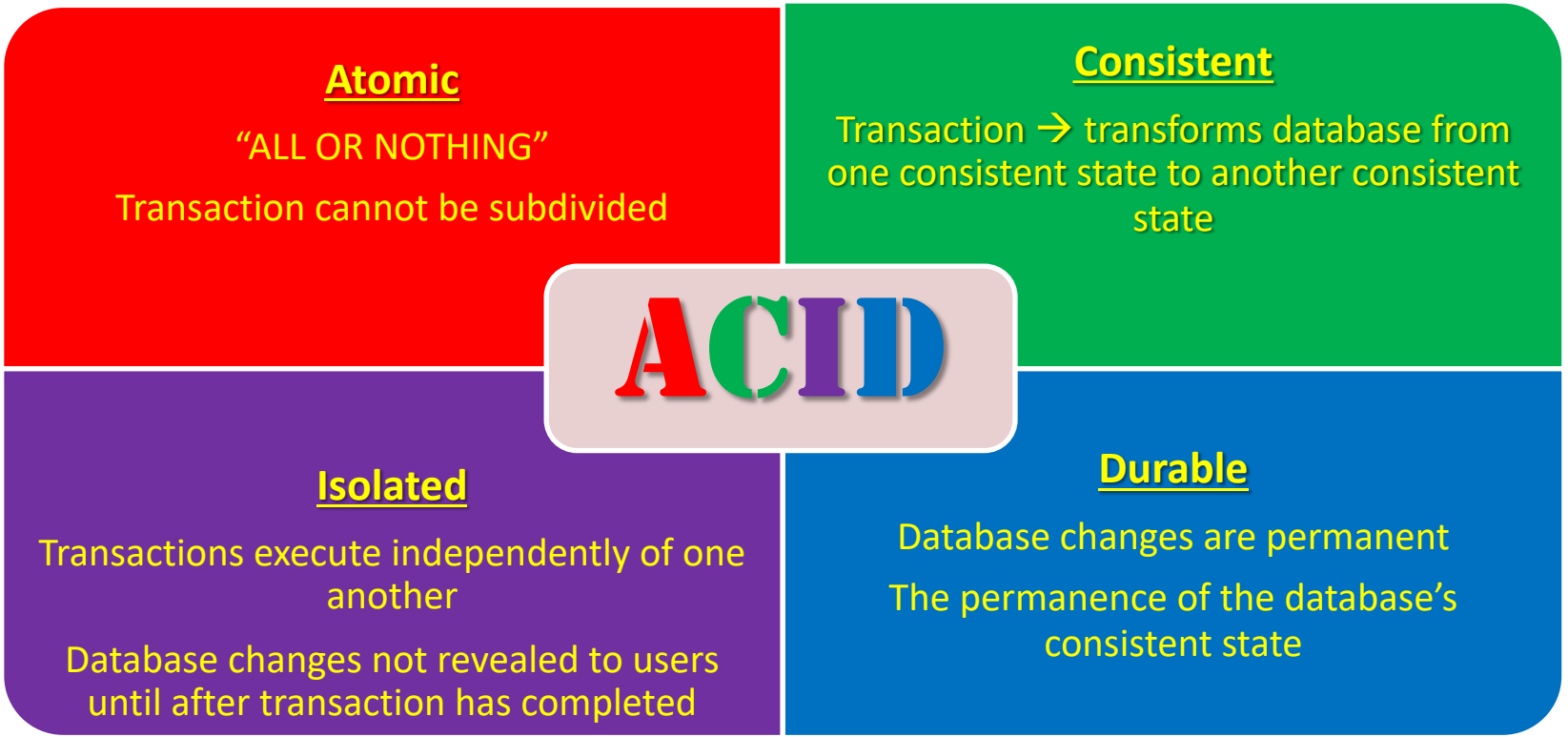


Database Systems: The Complete Book (2nd Edition)
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

ACID Concepts

Core Transaction Concept is ACID Properties

http://slideplayer.com/slide/9307681



ACID Properties (httpwww.tutorialspoint.com/dbms/dbms_transaction.htm)

A *transaction* is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Transaction Concept

BEGIN TRANSACTION {

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

COMMIT or ROLLBACK }

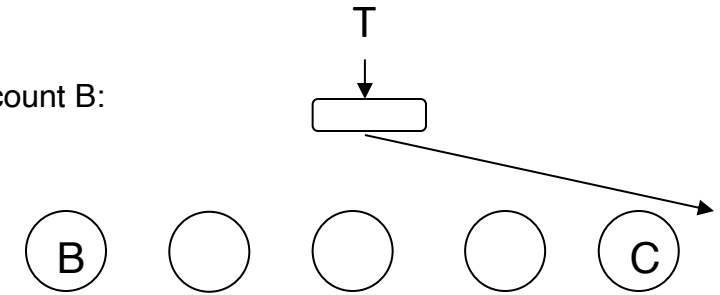
- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:

  BEGIN TRANSACTION {

  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

  COMMIT or ROLLBACK }
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer



- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

T1, T2

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1
T2

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A)
	read(B) print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Atomicity
Durability

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_account.overdraft_limit
3. ELSE
    1. Check that (source_count.balance-amount) >source_account.minimum_balance
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

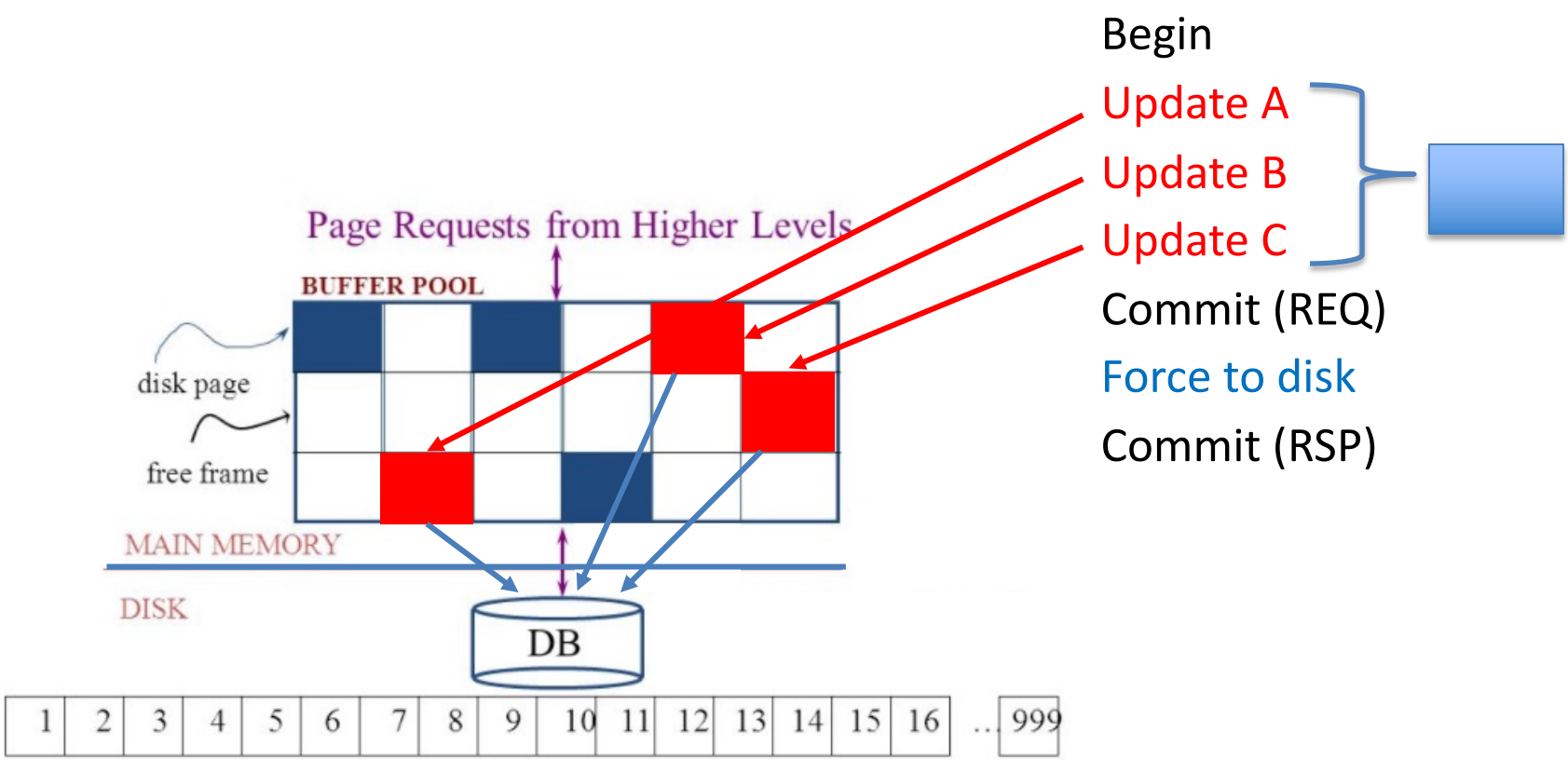
def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_ac
3. ELSE
    1. Check that (source_count.balance-amount) >source_a
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

- Transaction programs and databases are fast (milliseconds).
  What are the chances of the failure occurring in the wrong spot?
- Well, that doesn't really matter. If it happens,
  - Someone lost money and
  - There is no record off it. Someone is going to very upset.
- Even a small server can have thousands of concurrent transactions ➔
  - There will be corruptions because some transaction will be in the wrong place at the wrong time.
  - Unless we do something in the DBMS
  - Because HW and software inevitably fail
  - And sadly, SW is especially prone to failure when under load

Simplistic Approach



Simplistic Approach

There are several problems with the simplistic approach.

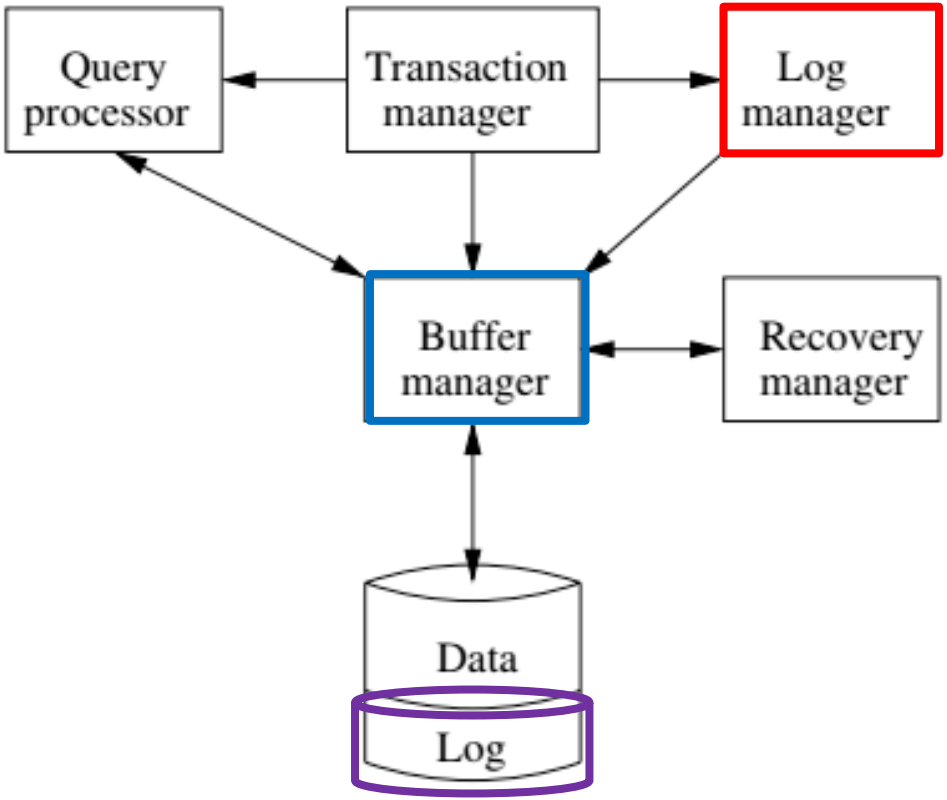
1. The approach does not solve the problem
    1. Some writes might succeed.
    2. Some might be interrupted by the failure, or require retry.
2. Writes may be random and scattered. N updates might
    1. Change a few bytes in N data frames
    2. A few bytes in M index frames

    Transaction rate becomes bottlenecked by write I/O rate, even though a relative small number of bytes change/transaction.
3. Written frames must be held in memory.
    1. Lots of transactions
    2. Randomly writing small pieces of lots of frames.
    3. Consumes lots of memory with pinned pages.
    4. Degrades the performance and optimization of the buffer.
        1. The optimal buffer replacement policy wants to hold frames that will be reused.
        2. Not frames that have been touched and never reused.

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query scheduling, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



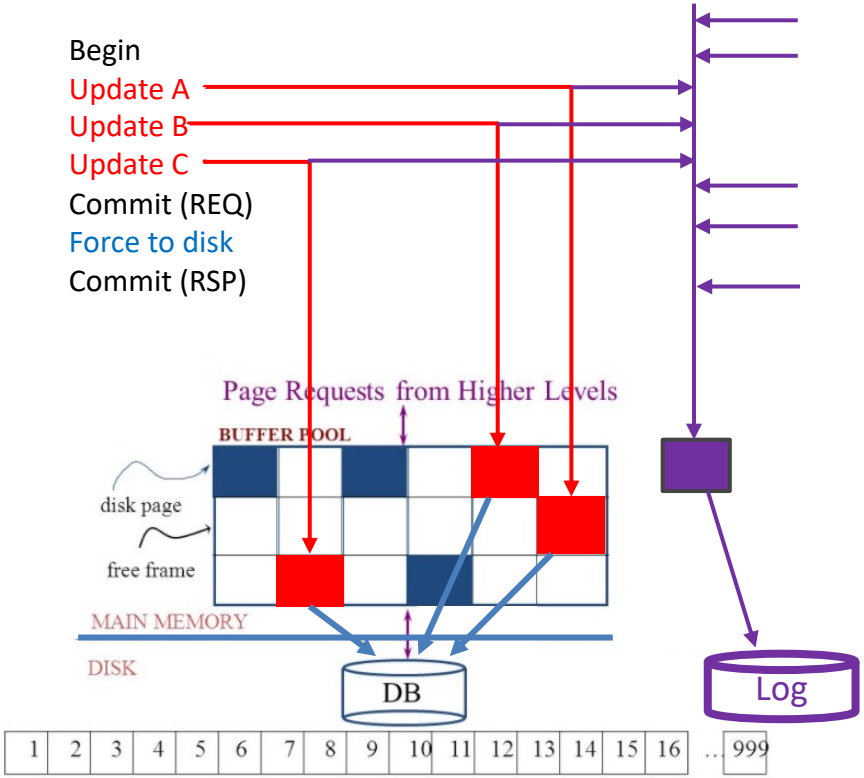
Garcia-Molina et al., p. 846

Logging

The DBMS logs every transaction event

- *Log Sequence Number* (LSN): A unique ID for a log record.
- *Prev LSN*: A link to their last log record.
- *Transaction ID number.*
- *Type*: Describes the type of database log record.
  - **Update Log Record**
    - *PageID*: A reference to the Page ID of the modified page.
    - *Length and Offset*: Length in bytes and offset of the page are usually included.
    - *Before and After Images* of records.
  - **Compensation Log Record**
  - **Commit Record**
  - **Abort Record Checkpoint Record**
  - **Completion Record** notes that all work has been done for this particular transaction.

Write Ahead Logging

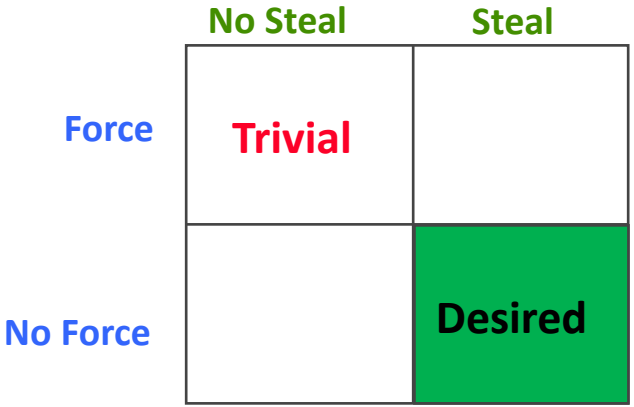


DBMS (Redo processing)

- Write log events from all transactions into a single log stream.
- Multiple events per page
- Forces (writes) log record on COMMIT/ABORT
  - Single block I/O records many updates
  - Versus multiple block I/Os, each recording a single change.
  - All of a transaction's updates recorded in one I/O versus many.
- If there is a failure
  - DBMS sequentially reads log.
  - Applies changes to modified pages that were not saved to disk.
  - Then resumes normal processing.

Write Ahead Logging

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor performance/caching performance
  - If yes, how can we ensure atomicity? Uncommitted updates on disk



DBMS (Undo processing)

- Enable steal policy to improve cache performance by
  - Avoiding lots of pinned pages
  - Unlikely to be reused soon.
- Before stealing
  - Force log record to disk.
  - Update log entry has data record
    - Before image
    - After image
- If there is a failure
  - DBMS sequentially reads log.
  - Undoes changes to
    - modified pages, uncommitted pages
    - That were saved to disk.
  - Then resumes normal processing.

ARIES recovery involves three passes

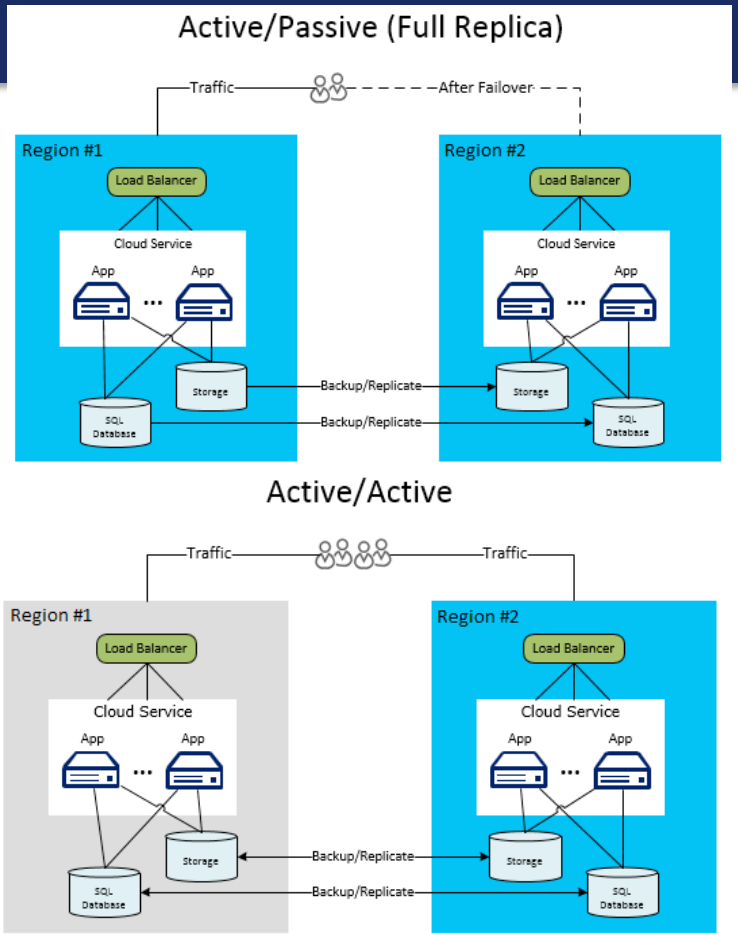
1. Analysis pass:
   - Determine which transactions to undo
   - Determine which pages were dirty (disk version not up to date) at time of
   - RedoLSN: LSN from which redo should start
2. Redo pass:
   - Repeats history, redoing all actions from RedoLSN
     (updated committed but not written changes to pages)
   - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
3. Undo pass:
   - Rolls back all incomplete transactions (with uncommitted pages written to disk).
   - Transactions whose abort was complete earlier are not undone

Durability

- Write changes to disk trivially achieves durability.
- DBMS engine uses write-ahead-logging to
  - Achieve durability
  - But with better performance through more efficient caching and I/O.
- Well, disks fail. How is that durable.
  - RAID and other solutions.
  - Disk subsystems, including entire RAID device, fail →
    - Duplex writes
    - To independent disk subsystems.
- Well, there are earthquakes, floods, etc.

Availability and Replication

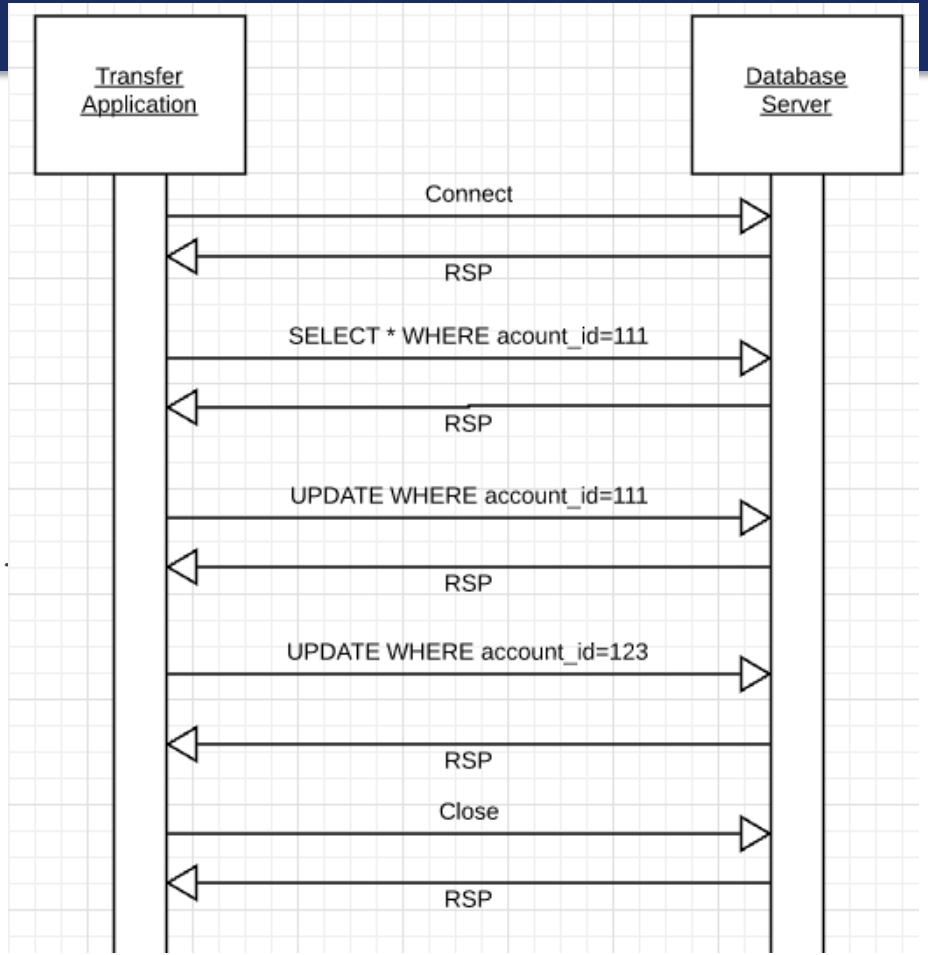
- There are two basic patterns
  - Active/Passive
    - All requests go to *master* during normal processing.
    - Updates are transactionally queued for processing at passive backup.
    - Failure of *master*
      - Routes subsequent requests to *backup.*
      - Backup must process and commit updates before accepting requests.
  - Active/Active
    - Both environments process requests.
    - Some form of distributed transaction commit required to synchronize updates on both copies.
- Multi-system communication to guarantee consistency is the foundation for tradeoffs in CAP.
  - The system can be CAP if and only iff
  - There are never any partitions or system failures
  - Which is unrealistic in cloud/Internet systems.



Isolation

Isolation

- Transfer $50 from
  - account_id=111 to
  - account_id=123
- Requires 3 SQL statements
  - SELECT from 111 to check balance >= $50
  - UPDATE account_id=111
  - UPDATE account_id=123
- There are some interesting scenarios
  - Two different programs read the balance ($51)
  - And decide removing $50 is OK.
- DB constraints can prevent the conflict from happening, but …
  - There are more complex scenarios that constraints do not prevent.
  - Not ALL databases support constraints.
  - The "correct" execution should be that
    - One transaction responds "insufficient funds"
    - Before attempting transfer instead of after attempting.



Isolation

- Consider two **simultaneous** transfer transactions T1 and T2.
- There are two equally ***correct*** executions
  1. T1 transfers, T2 responds “insufficient funds” and does not attempt transfer
  2. T2 transfers, T1 responds “insufficient funds” and does not attempt transfer
- Each correct simultaneous execution is equivalent to a serial (sequential) execution schedule
  - (1) Execute T1, Execute T2
  - (2) Execute T2, Execute T1
- NOTE:
  - We are focusing on ***correctness*** not
  - ***Fairness:***
    - We do not care which transaction was actually submitted first.
    - And probably do not know due to networking, etc.

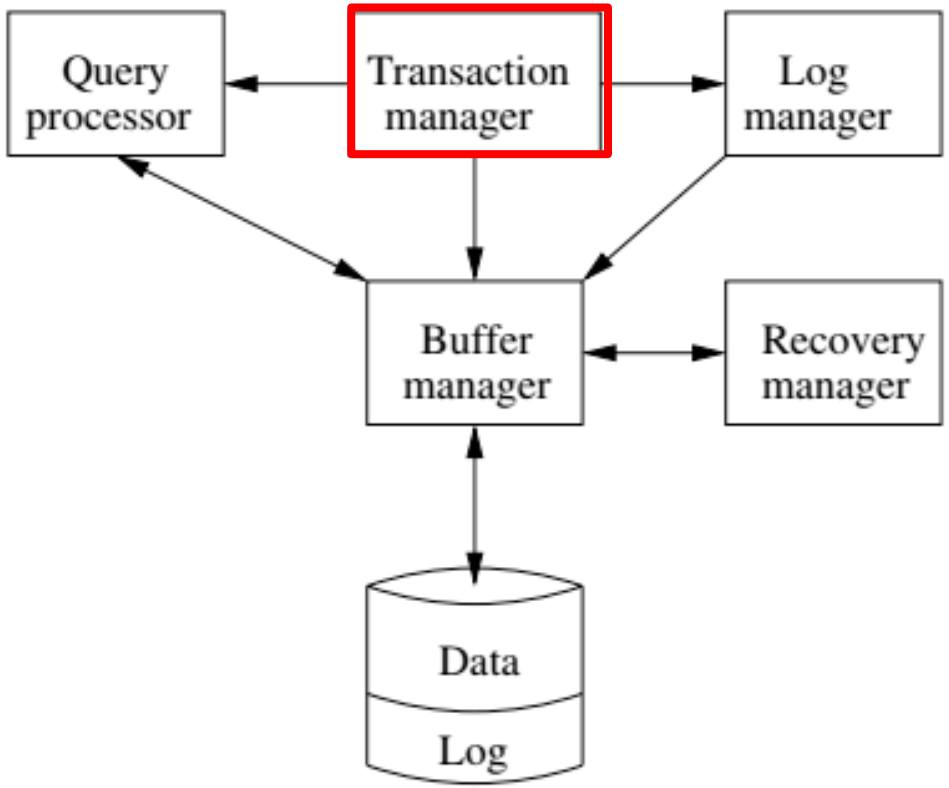
Serializability

"In concurrency control of databases,[1][2] transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems." (https://en.wikipedia.org/wiki/Serializability)

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query **scheduling**, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



Garcia-Molina et al., p. 846

Schedule

18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A.

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

Figure 18.2: Two transactions

Garcia-Molina et al.

Serializable

- Assume there are three
  - concurrently executing transactions
  - T1, T2 and T3
- The transaction manager
  - Enables concurrent execution
  - But schedules individual operations
  - To ensure that the final DB state
  - Is *equivalent* to one of the following schedules
    - T1, T2, T3
    - T1, T3, T2
    - T2, T1, T3
    - T2, T3, T1
    - T3, T1, T2
    - T3, T2, T1

  Concurrent execution was *serializable.*

18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

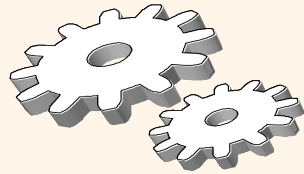
T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Figure 18.3: Serial schedule in which T_1 precedes T_2

Serializability (en.wikipedia.org/wiki/Serializability)

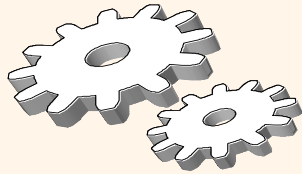
- **Serializability** is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction schedule (history). It relates to the *isolation* property of a database transaction.
- **Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.
- **The rationale behind serializability** is the following:
  - If each transaction is correct by itself, i.e., meets certain integrity conditions,
  - then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions):
    - "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete *isolation* between each other exists.
    - Any order of the transactions is legitimate, (…)
    - As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Lock-Based Concurrency Control



- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Aborting a Transaction



- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

MySQL (Locking) Isolation

13.3.6 SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
    ISOLATION LEVEL level
  | READ WRITE
  | READ ONLY

level:
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
```

Scope of Transaction Characteristics

You can set transaction characteristics globally, for the current session, or for the next transaction:

- With the `GLOBAL` keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.
- With the `SESSION` keyword, the statement applies to all subsequent transactions performed within the current session.
- Without any `SESSION` or `GLOBAL` keyword, the statement applies to the next (not started) transaction performed within the current session. Subsequent transactions revert to using the `SESSION` isolation level.

Isolation Levels (https://en.wikipedia.org/wiki/Isolation_(database_systems)

Not all transaction use cases require 2PL and serializable execution. Databases support a a set of levels.

- **Serializable**
  - With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the ***phantom reads*** phenomenon.
  - *The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.*
- **Repeatable reads**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. However, *range-locks* are not managed, so ***phantom reads*** can occur.
  - Write skew is possible at this isolation level, a phenomenon where two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.[3][4]
- **Read committed**
  - In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the ***non-repeatable reads phenomenon*** can occur in this isolation level). As in the previous level, *range-locks* are not managed.
  - Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.
- **Read uncommitted**
  - This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed, so one transaction may see *not-yet-committed* changes made by other transactions

In Databases, Cursors Define *Isolation*

- We have talked about ACID transactions

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

- Isolation
  - Determines what happens when two or more threads are manipulating the data at the same time.
  - And is defined relative to where cursors are and what they have touched.
  - Because the cursor movement determines *what you are reading or have read.*
- *But, … Cursors are client conversation state and cannot be used in REST.*

Relational Database Design
Normalization

Overview of Normalization

Features of Good Relational Designs

- Hypothetically, assume our schema originally had one relation that provided information about faculty and departments.
- Suppose we combine *instructor* and *department* into *in_dep,* which represents the natural join on the relations *instructor* and *department*

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

X → Y
X = (dept_name)
Y = (building, budget)

Pix Physics
Pix Physics

Piy (Watson, 70000)

- There is repetition of information
- Need to use null values (if we add a new department with no instructors)

Decomposition

- The only way to avoid the repetition-of-information problem in the i*n_dep* schema is to decompose it into two schemas – instructor and *department* schemas.
- Not all decompositions are good. Suppose we decompose

  employee(ID, name, street, city, salary)

  into

  employee1 (*ID*, *name*)

  employee2 (*name*, *street, city, salary*)

  The problem arises when we have two employees with the same name
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

Lossy Decomposition

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

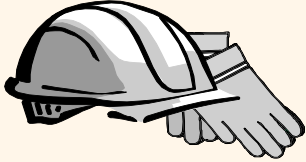
ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

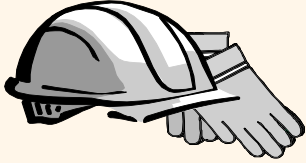
natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

The Evils of Redundancy



- *Redundancy* is at the root of several problems associated with relational schemas:
  - redundant storage, insert/delete/update anomalies
- Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.
- Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).
- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?



Example (Contd.)

Wages

R	W
8	10
5	7

Hourly_Emps2

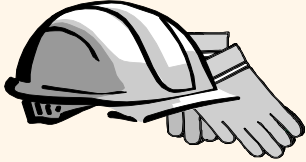
S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

- Problems due to R → W :
  - *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
  - *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
  - *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Will 2 smaller tables be better?

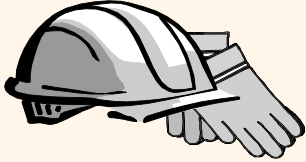
S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Functional Dependencies (FDs)



- A functional dependency X→ Y holds over relation R if, for every allowable instance *r* of R:
  - $t1 \in r$, $t2 \in r$, $\pi_X(t1) = \pi_X(t2)$ implies $\pi_Y(t1) = \pi_Y(t2)$
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)
- An FD is a statement about *all* allowable relations.
  - Must be identified based on semantics of application.
  - Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R!
- K is a candidate key for R means that K→ R
  - However, K→ R does not require K to be *minimal*!

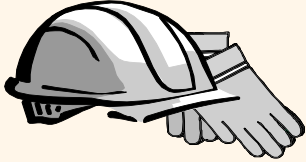
Example: Constraints on Entity Set



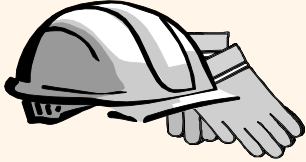
- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (*ssn*, *name*, *lot*, *rating*, *hrly_wages*, *hrs_worked*)
- *Notation*: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the *set* of attributes {S,N,L,R,W,H}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)
- Some FDs on Hourly_Emps:
  - *ssn* is the key: S → SNLRWH
  - *rating* determines *hrly_wages*: R → W

- (uni, email, last_name, first_name)
  uni and email are candidate (not NULL, unique)
  - Uni → email
  - Email → (last_name, first_name)
  - ➔ Uni → (last_name, first_name)
- Person:(uni, email, last_name, first_name),
  ContactInfo: (email, phone_number, dorm_name)
  - Uni → email
  - Email → phone_number, dorm_name
  - Uni → phone_number, dorm_name

Reasoning About FDs



- Given some FDs, we can usually infer additional FDs:
  - *ssn* → *did*, *did* → *lot* implies *ssn* → *lot*
- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - F^+ = *closure of F* is the set of all FDs that are implied by *F*.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - *Reflexivity*: If $X \subseteq Y$, then $Y \rightarrow X$
  - *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These are *sound* and *complete* inference rules for FDs!



Lossless Decomposition

- Let *R* be a relation schema and let R_1 and R_2 form a decomposition of R.
  - That is R = R_1 U R_2
  - *Note: This notation is confusing. This is a statement about the schema, not about the data in relations.*
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas R_1 U R_2
- Formally,

$$\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

Normalization Theory

- Decide whether a particular relation R is in "good" form.
- In the case that a relation R is not in "good" form, decompose it into set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
  - Students and instructors are uniquely identified by their ID.
  - Each student and instructor has only one name.
  - Each instructor and student is (primarily) associated with only one department.
  - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
- A legal instance of a database is one where all the relation instances are legal instances
- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key.*

Functional Dependencies Definition

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r.

1	4
1	5
3	7

- On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does **NOT** hold,

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
  - etc.
- The set of **all** functional dependencies logically implied by F is the **closure** of F.
- We denote the *closure* of F by F^+.

Keys and Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  in_dep (*ID, name, salary, dept_name, building, budget*).

  We expect these functional dependencies to hold:

  dept_name → *building*

  ID → *building*

  but would not expect the following to hold:

  dept_name → *salary*

DFF Note:
- In the current data: ID → dept_name → building
- But
- In the schema, dept_name may be NULL..

Use of Functional Dependencies

- We use functional dependencies to:
  - To test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
  - To specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy
    name $\rightarrow$ *ID.*

Normal Forms

Boyce-Codd Normal Form

- A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

  $$\alpha \rightarrow \beta$$

  where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R

DFF Note:
- The theoretical treatment is no conveying the practical intent.
- If α is α superkey, I can set a primary key/unique constraint on α.
- Consider tables with address info in the rows.

Decomposing a Schema into BCNF

- Let R be a schema R that is not in BCNF. Let $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.
- We decompose R into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example of *in_dep*,
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by
  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, dept_name, salary*)

DFF Note – again this is baffling
- R = (id, name_last, name_first, street, city, state, zipcode)
- $\alpha \rightarrow \beta$ means zipcode → (city, state)
- $(\alpha \cup \beta)$ = (zipcode, city, state), which we call Address
- $R - (\beta - \alpha)$). = R - β + α = (id, name_last, name_first, zipcode), which we call Person
- Setting primary key ID in Address and zipcode on Address preserves the dependency and is lossless.

Note:
- This is an example only.
- Zip code does not imply city or state in real world.

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema:

  dept_advisor(s_ID, i_ID, department_name)
- With function dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$
- *dept_advisor* is not in BCNF
  - *i_ID* is not a superkey.
- Any decomposition of *dept_advisor* will not include all the attributes in

  $s\_ID, dept\_name \rightarrow i\_ID$
- Thus, the composition is NOT be dependency preserving

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

  $$\alpha \rightarrow \beta \text{ in } F^+$$

  at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

    (**NOTE**: each attribute may be in a different candidate key)
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

3NF Example

- Consider a schema:

  dept_advisor(s_ID, i_ID, dept_name)

- With function dependencies:

  i_ID $\rightarrow$ *dept_name*

  s_ID, dept_name $\rightarrow$ *i_ID*

- Two candidate keys = {*s_ID, dept_name*}, {*s_ID, i_ID* }
- We have seen before that *dept_advisor* is not in BCNF
- *R,* however, is in 3NF
  - *s_ID, dept_name* is a superkey
  - *i_ID* $\rightarrow$ *dept_name and i_ID* is NOT a superkey, but:
    - { *dept_name*} – {*i_ID* } = {*dept_name* } and
    - *dept_name* is contained in a candidate key

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.
- Disadvantages to 3NF.
  - We may have to use null values to represent some of the possible meaningful relationships among data items.
  - There is the problem of repetition of information.

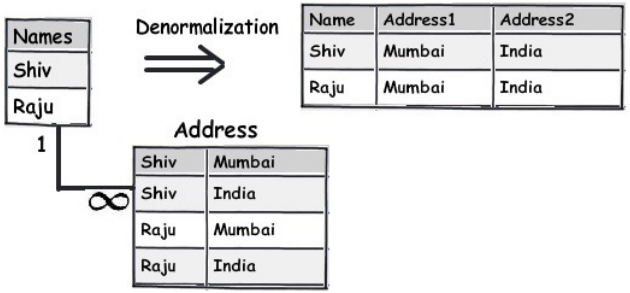
Goals of Normalization

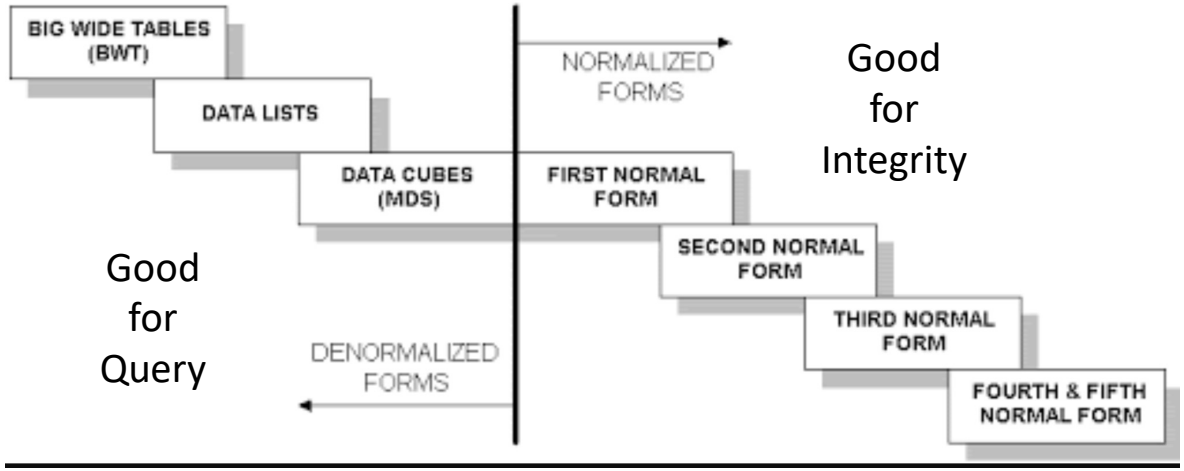
- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, need to decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that:
  - Each relation scheme is in good form
  - The decomposition is a lossless decomposition
  - Preferably, the decomposition should be dependency preserving.

Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined a *course* $\bowtie$ *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Wide Flat Tables





- Improve query performance by precomputing and saving:
  - JOINs
  - Aggregation
  - Derived/computed columns
- One of the primary strength of the relational model is maintaining "integrity" when applications create, update and delete data. This relies on:
  - The core capabilities of the relational model, e.g. constraints.
  - A well-design database (We will cover a formal definition – "normalization" in more detail later.
- Data models that are well designed for integrity are very bad for read only analysis queries.
  We will build and analyze wide flat tables as part of the analysis tasks in HW3, HW4 as projects.

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS 101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases)

First Normal Form (Cont.)

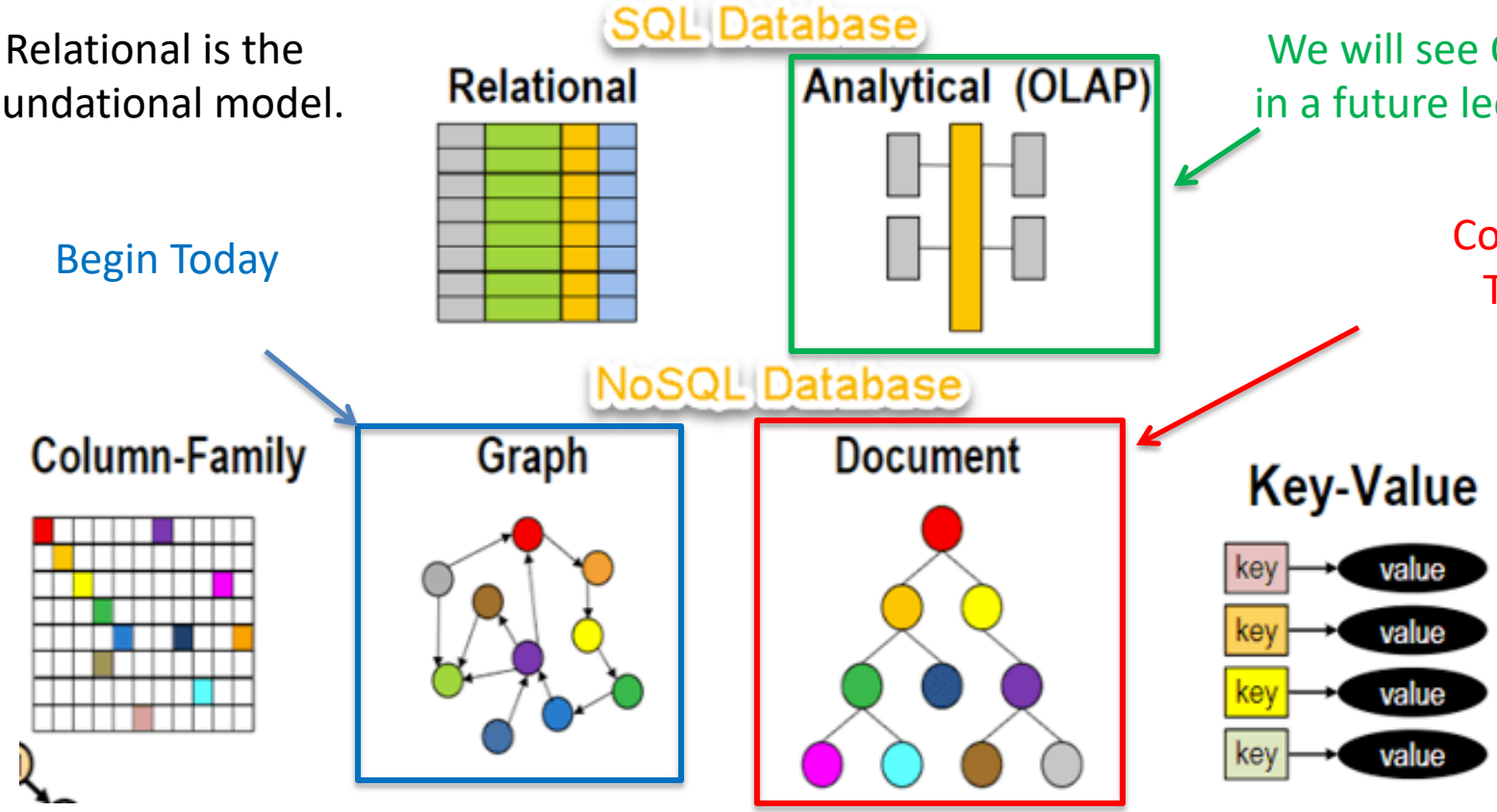
- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

NoSQL (3)

Reminder

Simplistic Classification (https://medium.com/swlh/4-types-of-nosql-databases-d88ad21f7d3b)

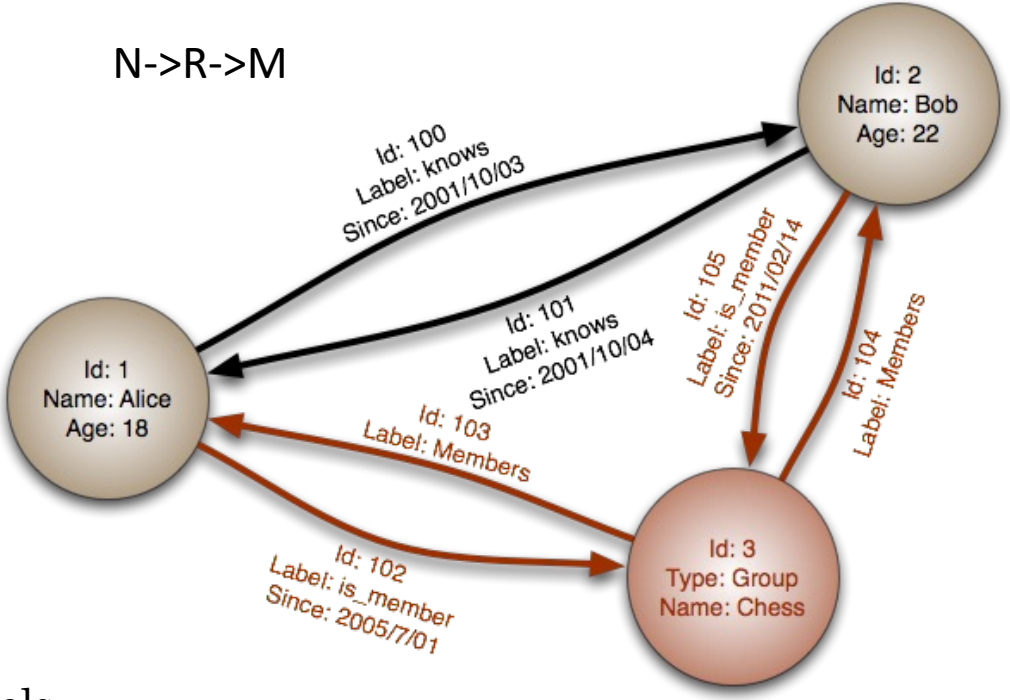
Relational is the foundational model.



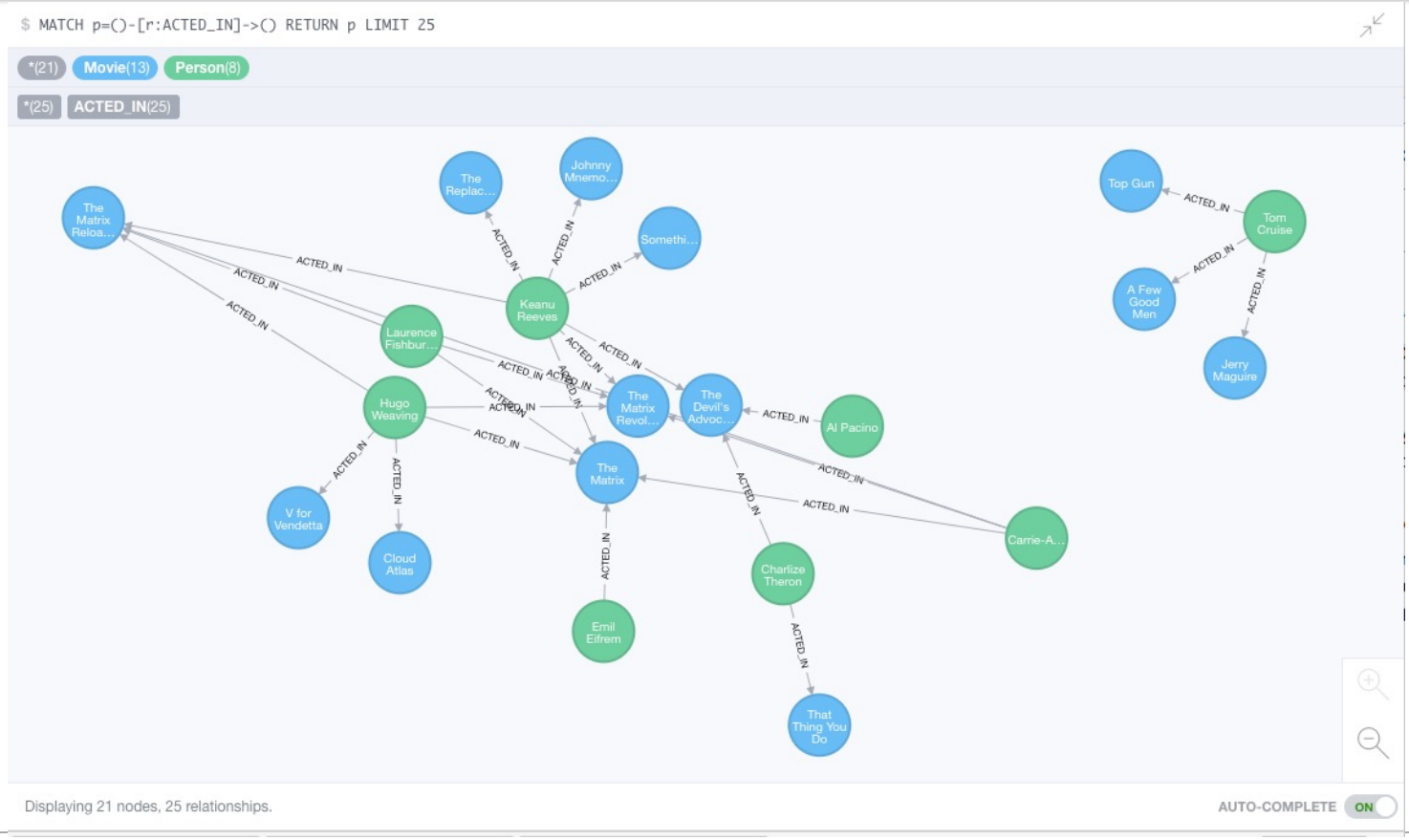
Graph Databases
Neo4j

Graph Database

- Exactly what it sounds like
- Two core types
  - Node
  - Edge (link)
- Nodes and Edges have
  - Label(s) = “Kind”
  - Properties (free form)
- Query is of the form
  - p1(n)-p2(e)-p3(m)
  - n, m are nodes; e is an edge
  - p1, p2, p3 are predicates on labels



Neo4J Graph Query



Why Graph Databases?

- Schema Less and Efficient storage of Semi Structured Information
- No O/R mismatch – very natural to map a graph to an Object Oriented language like Ruby.
- Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- Very natural to express graph related problem with traversals (recommendation engine, find shortest parth etc..)
- Seamless integration with various existing programming languages.
- ACID Transaction with rollbacks support.
- Whiteboard friendly – you use the language of node,properties and relationship to describe your domain (instead of e.g. UML) and there is no need to have a complicated O/R mapping tool to implement it in your database. You can say that Neo4j is "Whiteboard friendly" !(http://video.neo4j.org/JHU6F/live-graph-session-how-allison-knows-james/)

Social Network "path exists" Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

Graph databases are

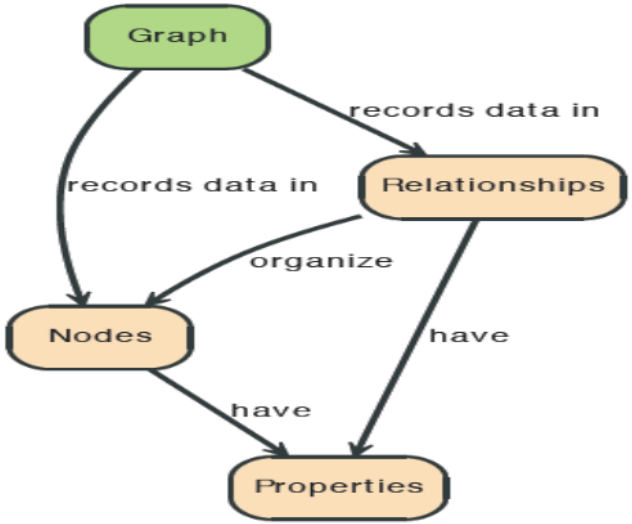
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

What are graphs good for?

- Recommendations
- Business intelligence
- Social computing
- Geospatial
- Systems management
- Web of things
- Genealogy
- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- Indexing your *slow* RDBMS
- And much more!

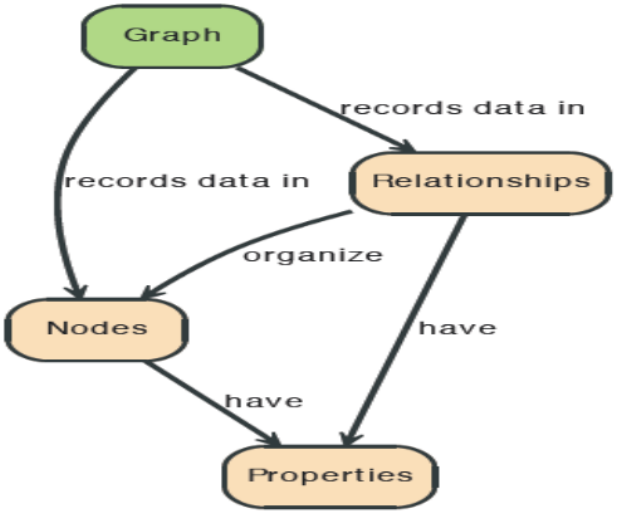
Graphs

- "A Graph —records data in→ Nodes —which have→ Properties"



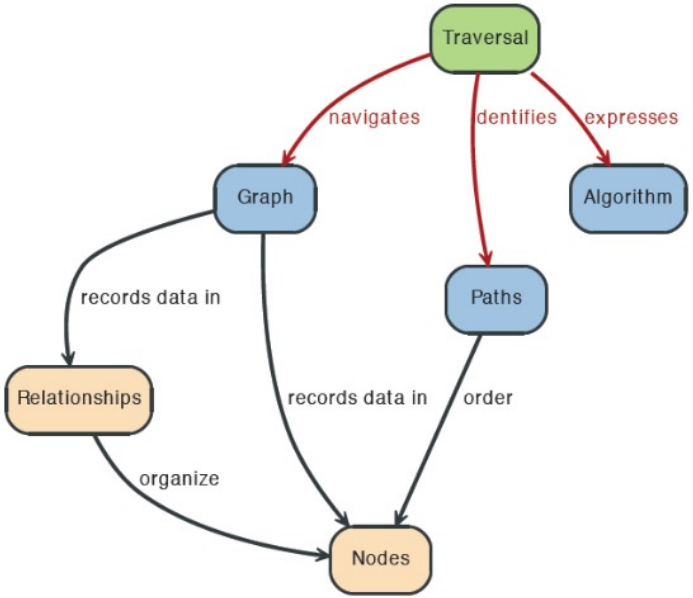
Graphs

- “Nodes —are organized by→ Relationships — which also have→ Properties”



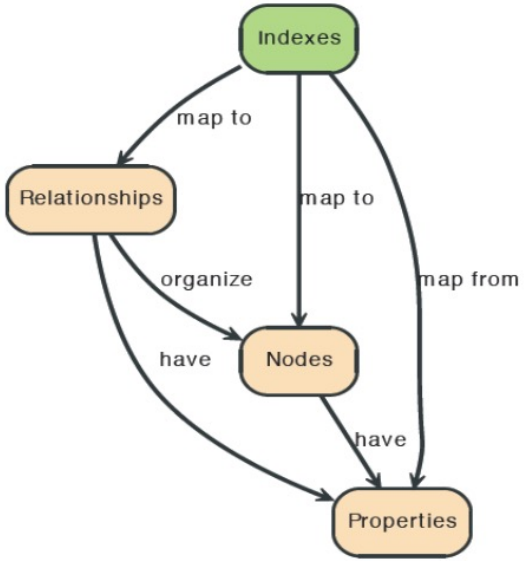
Query a graph with Traversal

- "A Traversal —navigates→ a Graph; it — identifies→ Paths —which order→ Nodes"

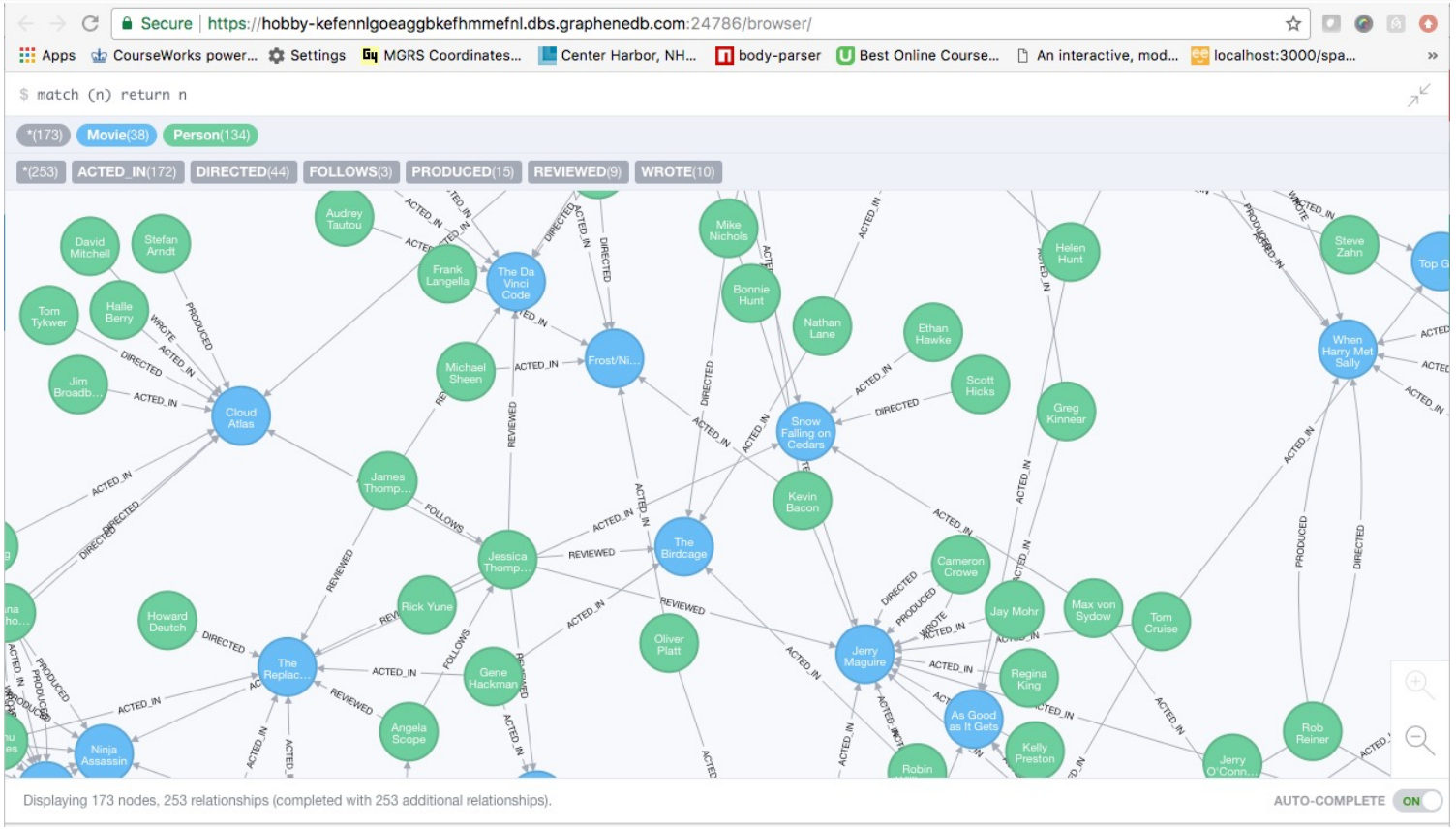


Indexes

- "An Index —maps from→ Properties —to either→ Nodes or Relationships"

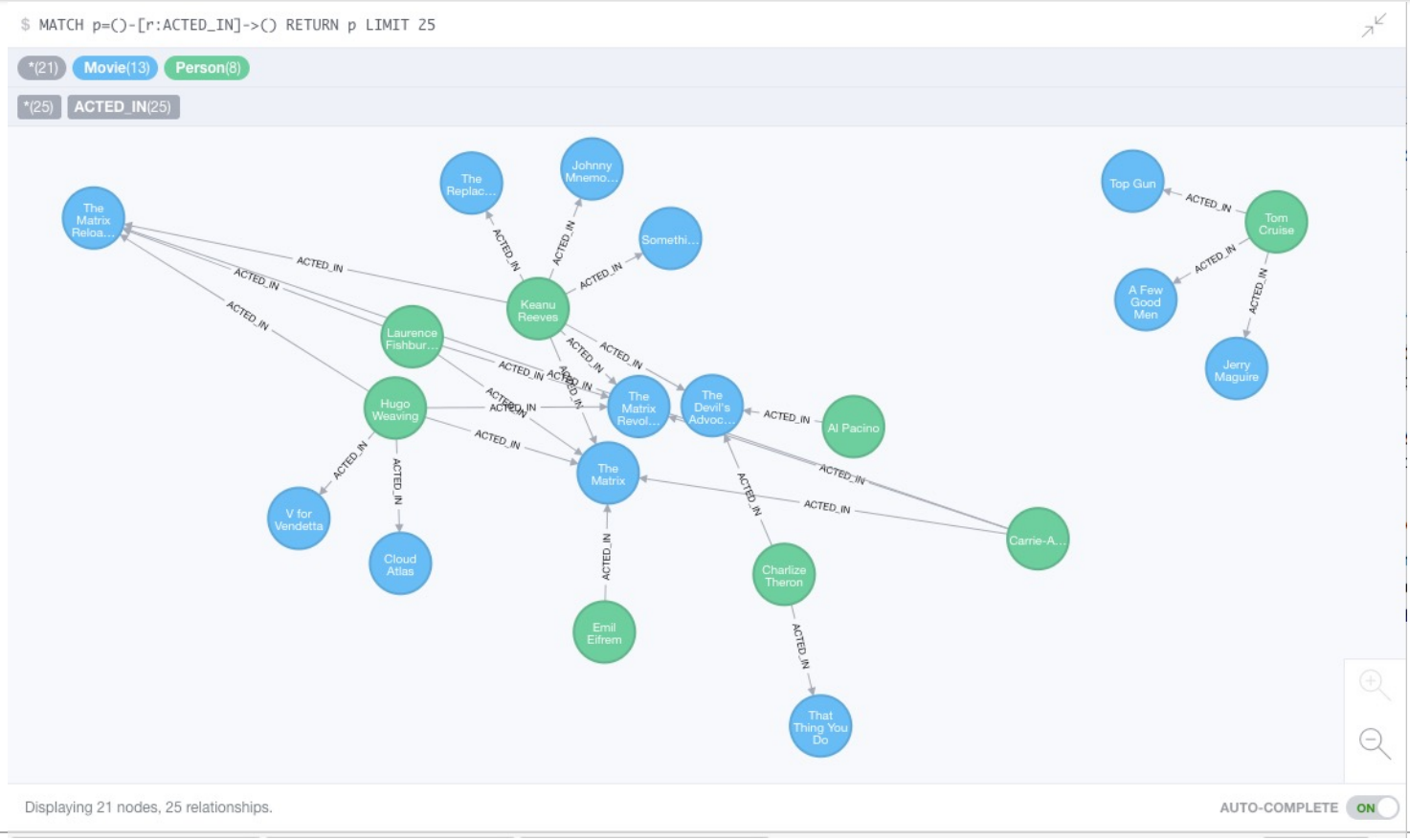


A Graph Database (Sample)



Neo4J Graph Query

Who acted in which movies?



Big Deal. That is just a JOIN.

- Yup. But that is simple.
- Try writing the queries below in SQL.

The Movie Graph

Recommend

Let's recommend new co-actors for Tom Hanks. A basic recommendation approach is to find connections past an immediate neighborhood which are themselves well connected.

For Tom Hanks, that means:

1. Find actors that Tom Hanks hasn't yet worked with, but his co-actors have.
2. Find someone who can introduce Tom to his potential co-actor.

Extend Tom Hanks co-actors, to find co-co-actors who haven't work with Tom Hanks...

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Find someone to introduce Tom Hanks to Tom Cruise

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

Recommend

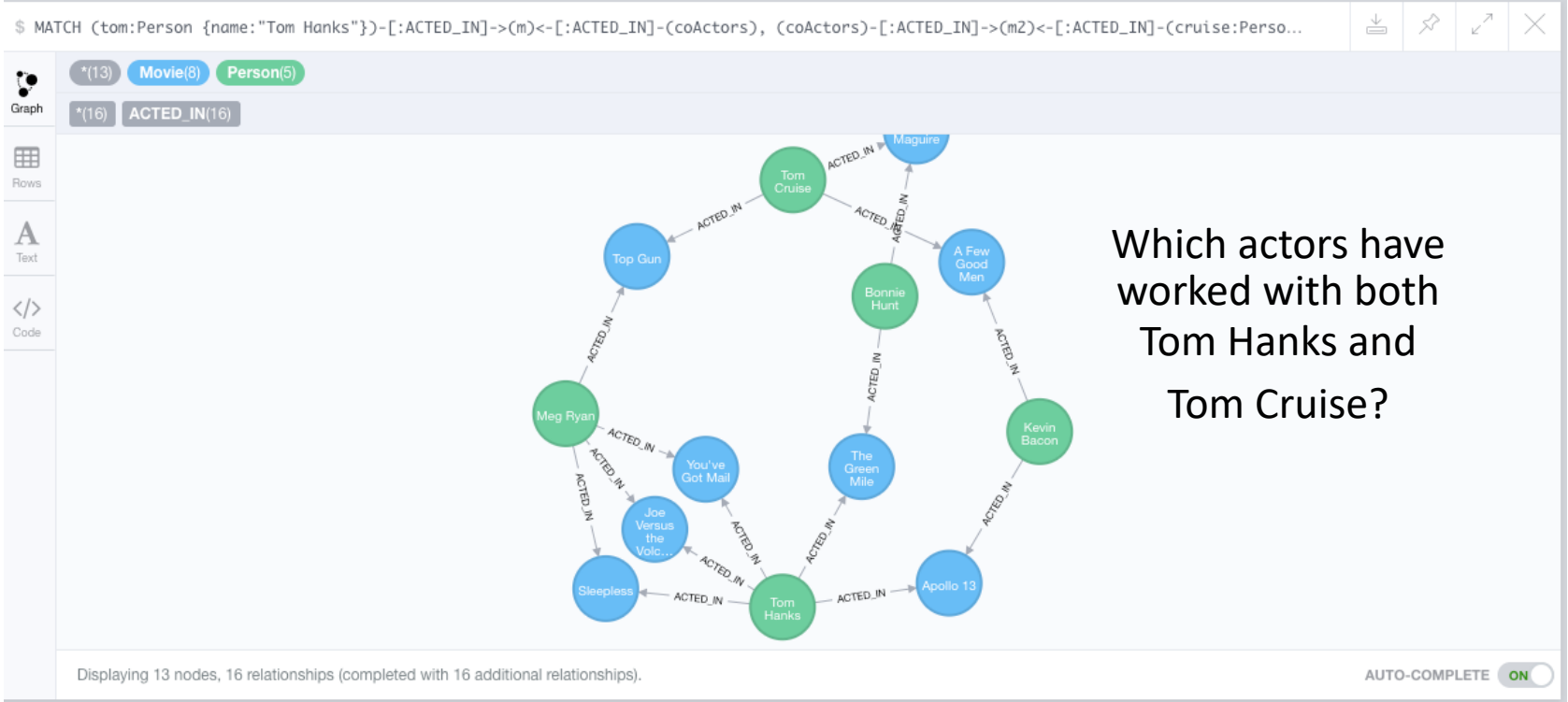
```
1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
2       (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
3 WHERE NOT (tom)-[:ACTED_IN]->(m2)
4 RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

$ MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors), (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors) ...

Rows | Text | Code

Recommended	Strength
Tom Cruise	5
Zach Grenier	5
Helen Hunt	4
Cuba Gooding Jr.	4
Keanu Reeves	4
Tom Skerritt	3
Carrie-Anne Moss	3
Val Kilmer	3
Bruno Kirby	3
Philip Seymour Hoffman	3
Billy Crystal	3
Carrie Fisher	3

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

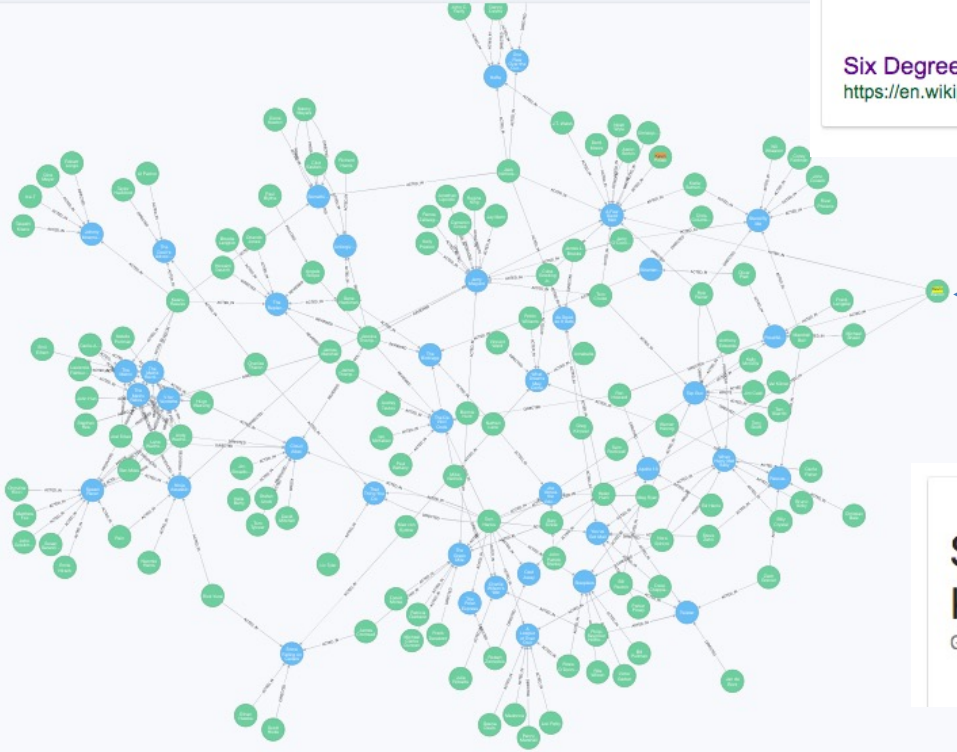


Which actors have worked with both Tom Hanks and Tom Cruise?

```
$ MATCH (s:Person { name: 'Kevin Bacon' })-[*0..6]-(m) return s,m
```

*(171) Movie(38) Person(133)

*(253) ACTED_IN(172) DIRECTED(44) FOLLOWS(3) PRODUCED(15) REVIEWED(9) WROTE(10)

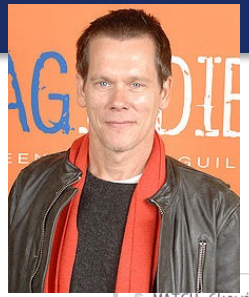


Six **Degrees of Kevin Bacon** is a parlour game based on the "six **degrees** of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin Bacon**.

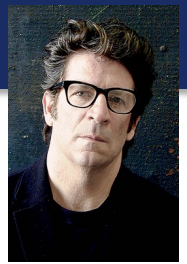
Six Degrees of Kevin Bacon - Wikipedia
https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

About this result Feedback

Six Degrees of Kevin Bacon
Game



How do you get from Kevin Bacon to Robert Longo?

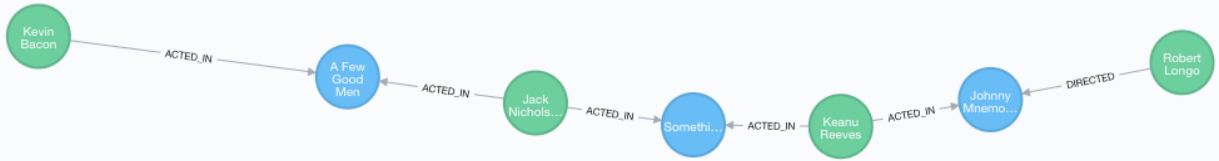


```
$ MATCH (kevin:Person { name: 'Kevin Bacon' }),(robert:Person { name: 'Robert Longo' }), p = shortestPath((kevin)-[*..15]-(robert)) RETURN p
```

Graph | Rows | Text | Code

*(7) Movie(3) Person(4)

*(6) ACTED_IN(5) DIRECTED(1)



Backup

Cloud Concepts – One Perspective

Categorizing and Comparing the Cloud Landscape

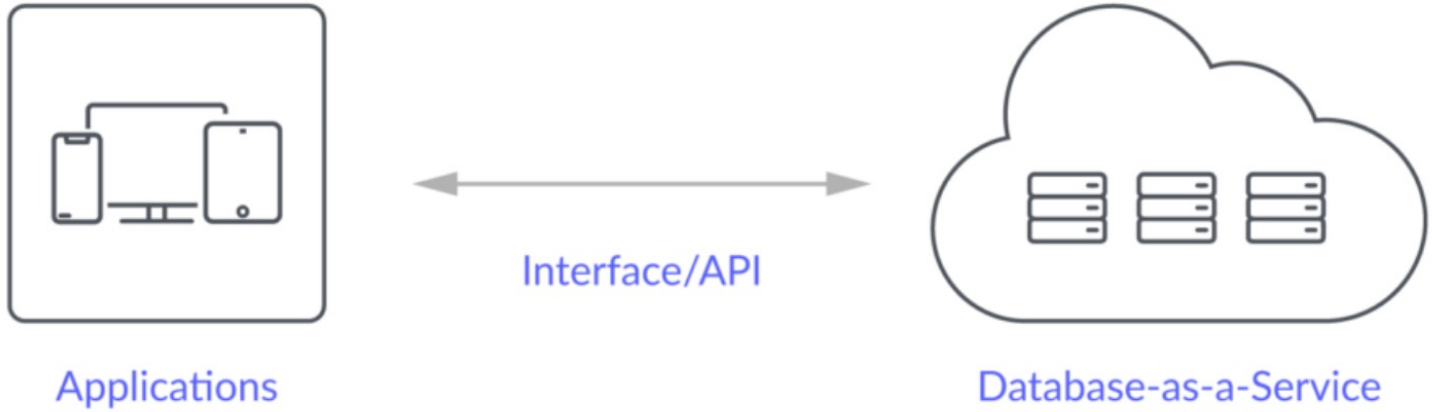
http://www.theenterprisearchitect.eu/blog/2013/10/12/the-cloud-landscape-described-categorized-and-compared/

6	**SaaS**	Applications			*End-users*
5	**App Services**	App Services	Communication and Social Services	Data-as-a-Service	*Citizen Developers*
4	**Model-Driven PaaS**	Model-Driven aPaaS, bpmPaaS	Model-Driven iPaaS	Data Analytics, baPaaS	*Rapid Developers*
3	**PaaS**	aPaaS	iPaaS	dbPaaS	*Developers / Coders*
2	**Foundational PaaS**	Application Containers	Routing, Messaging, Orchestration	Object Storage	*DevOps*
1	**Software-Defined Datacenter**	Virtual Machines	Software-Defined Networking (SDN), NFV	Software-Defined Storage (SDS), Block Storage	*Infrastructure Engineers*
0	**Hardware**	Servers	Switches, Routers	Storage	
		Compute	Communicate	Store	

Database-as-a-Service

“A cloud database is a database that typically runs on a cloud computing platform and access to the database is provided as-a-service. There are two common deployment models: users can run databases on the cloud independently, using a virtual machine image, or they can purchase access to a database service, maintained by a cloud database provider. Of the databases available on the cloud, some are SQL-based and some use a NoSQL data model.

Database services take care of scalability and high availability of the database. Database services make the underlying software-stack transparent to the user.” (Wikipedia)



Cloud Computing Layers