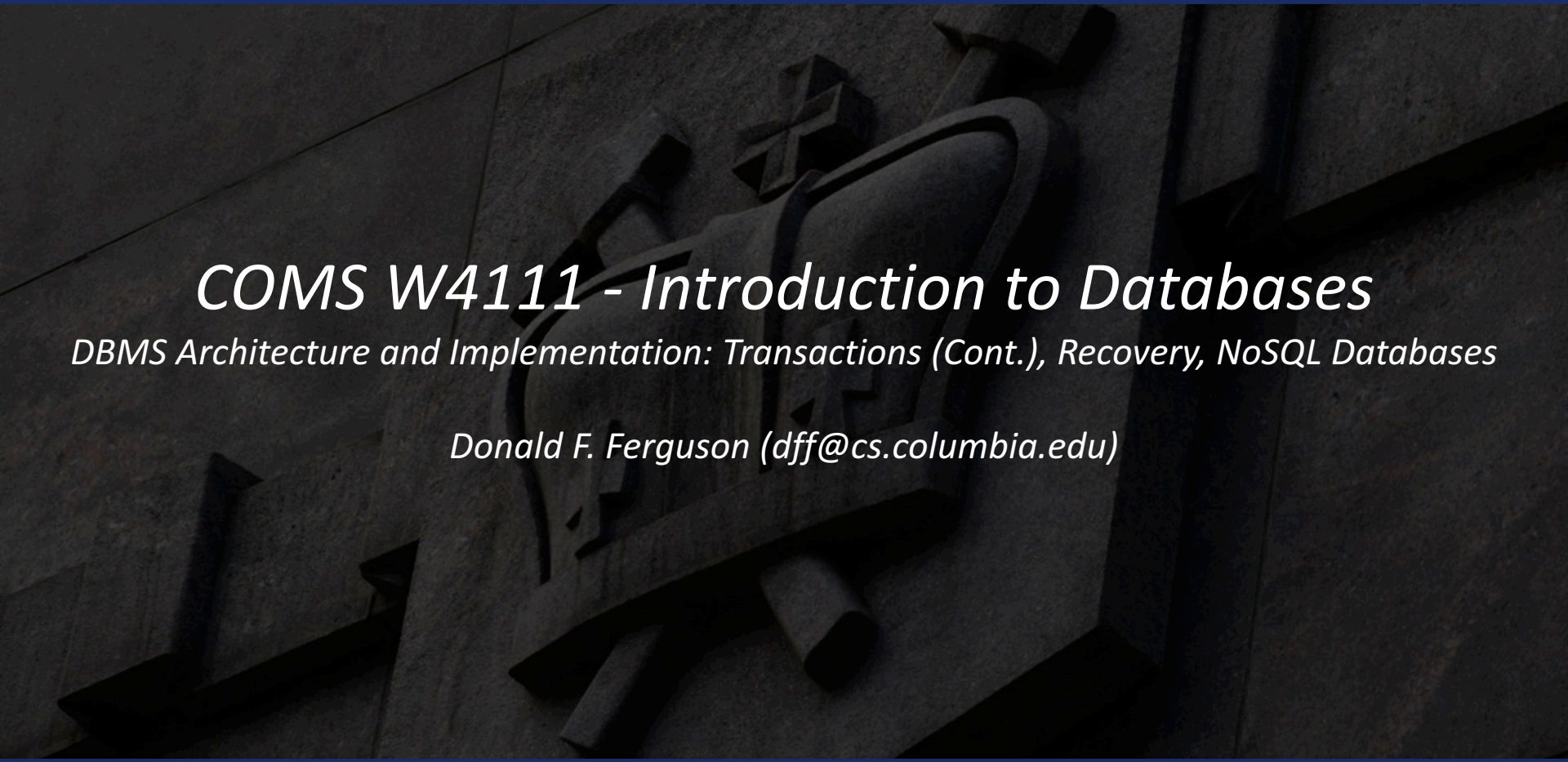


COMS W4111 - Introduction to Databases

DBMS Architecture and Implementation: Transactions (Cont.), Recovery, NoSQL Databases

Donald F. Ferguson (dff@cs.columbia.edu)

Contents

Contents

- Completing transactions and recovery:
  - I made a few mistakes (around isolation) when using the chalkboard to “clarify” slides.
  - Examples in this space can be tricky and detailed, and ”winging it” often results in my making mistakes.
  - Specific topics:
    - Schedules and serializability.
    - Locking, deadlocks.
    - Optimistic transactions.
    - ETags.
- NoSQL Databases:
  - Concepts, classification, motivation.
  - CAP Theorem
  - Eventual Consistency
  - Three examples: Redis, Neo4J, DynamoDB
  - Usage scenarios for each.

Transactions/Recovery: Correction and Continued

Isolation Concept

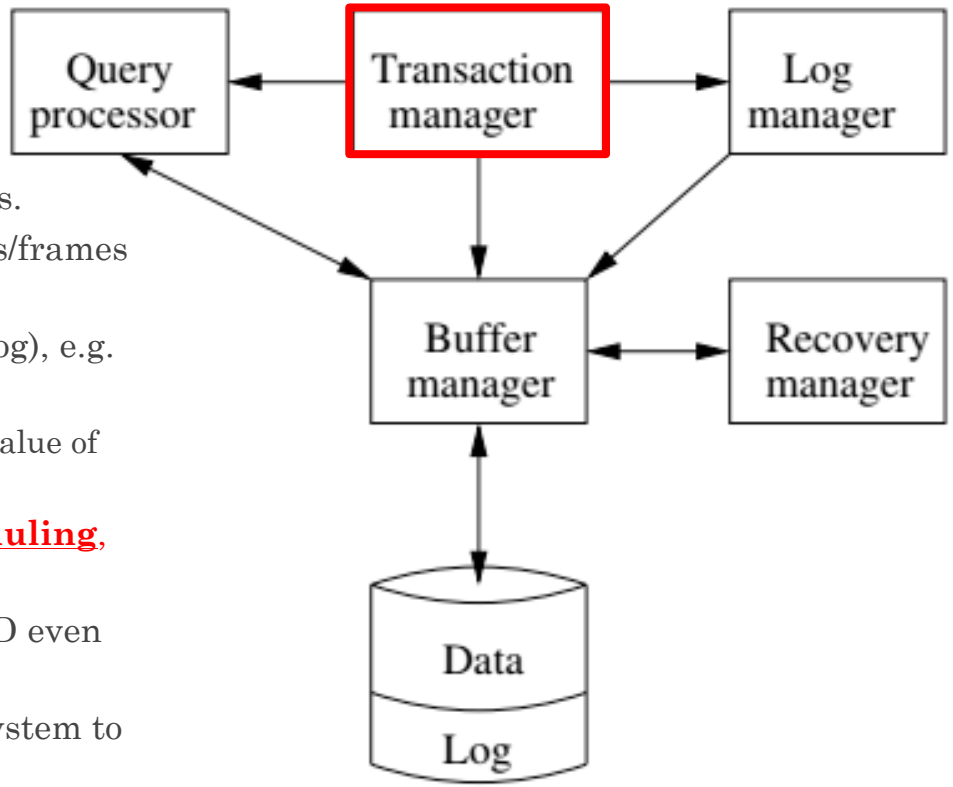
Serializability

"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

- Disks implement durability.
- All programs read/write data by accessing
  - The relevant bytes
  - Of a block
  - In a buffer, memory frame.

Garcia-Molina et al.

Serializable

- For concurrently executing transactions.
  - Assume there are three
  - concurrently executing transactions
  - T1, T2 and T3
- The transaction manager
  - Enables concurrent execution
  - But schedules individual operations in transactions.
  - 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*.

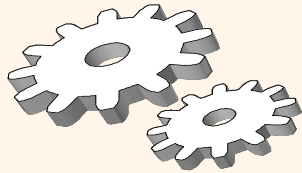
18.1.2 Serial Schedules

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

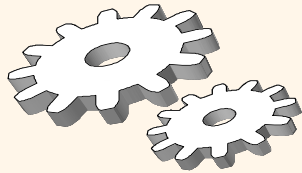
Scheduling Transactions



- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule:* A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

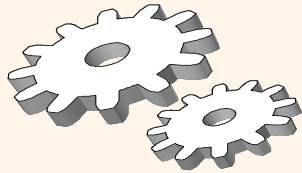


- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:              R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:  R(A),                  R(A), W(A), C
T2:       R(A), W(A), C
```



Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),          W(B), C
T2:       W(A), W(B), C
```

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.

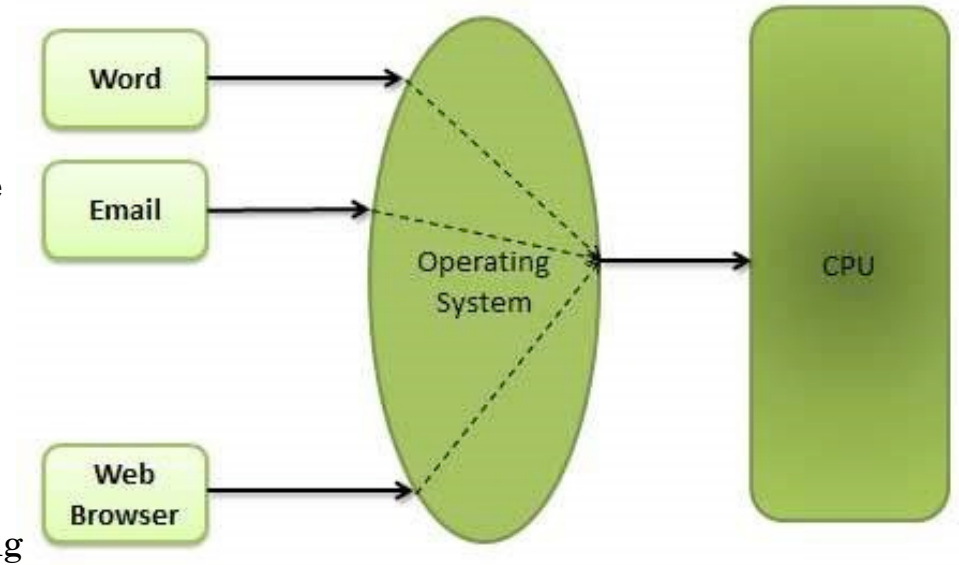
Locking

Locking and Concurrency

- Application code requires a “processor/CPU” to execute.
- The operating system (or execution server) allocates the CPU to a program/subprogram that is ”ready to run,” e.g.
  - Not waiting for an I/O to complete.
  - Not waiting for user input.
  - etc.
- The operating system suspends a program and reassigns a CPU when the application performs a blocking operation, e.g.
  - Make a remote function call.
  - A disk I/O is necessary.
  - The program must wait for a physical or logical resource, e.g. “a lock.”
- Acquiring a database lock is (potentially) a *blocking operation* that suspends the application until the lock request is satisfied.

Multitasking

In computing, **multitasking** is the concurrent execution of multiple tasks (also known as processes) over a certain period of time. New tasks can interrupt already started ones before they finish, instead of waiting for them to end. As a result, a computer executes segments of multiple tasks in an interleaved manner, while the tasks share common processing resources such as central processing units (CPUs) and main memory. Multitasking automatically interrupts the running program, saving its state (partial results, memory contents and computer register contents) and loading the saved state of another program and transferring control to it. This "context switch" may be initiated at fixed time intervals (pre-emptive multitasking), or the running program may be coded to signal to the supervisory software when it can be interrupted (cooperative multitasking).



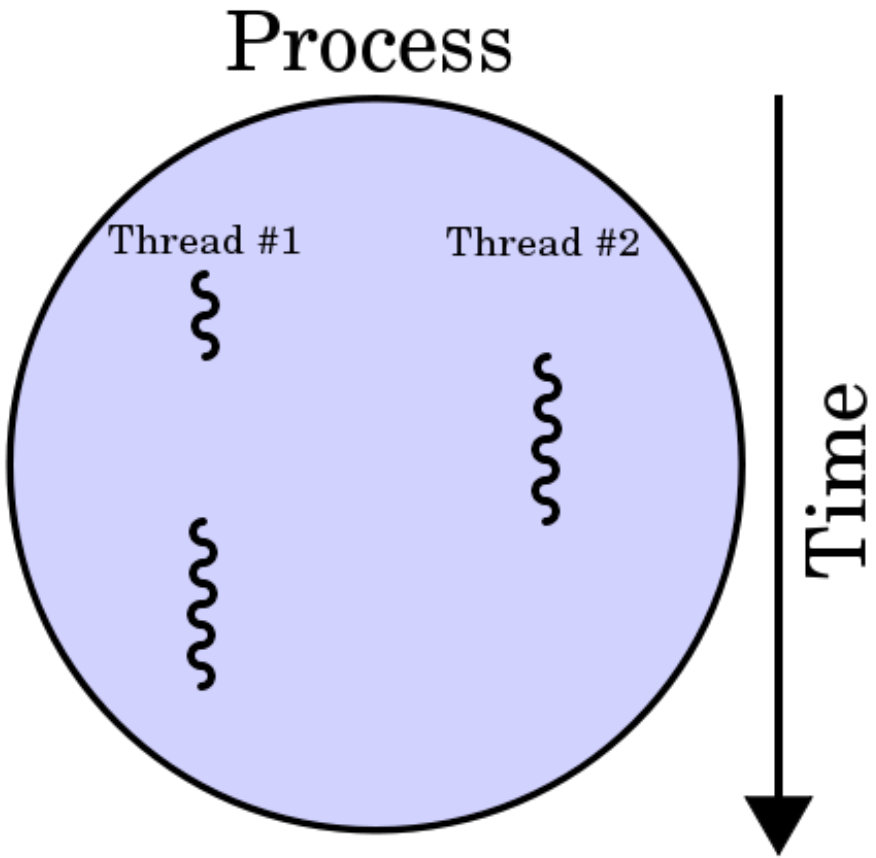
Operating system allocates the CPU to one "ready to run" process at a time.

The OS performs a context switch periodically, or when the process performs a non-CPU operation, e.g. I/O, wait for a message.

Multi-threading

In computer architecture, **multithreading** is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to execute multiple processes or threadsconcurrently, supported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the processes and threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

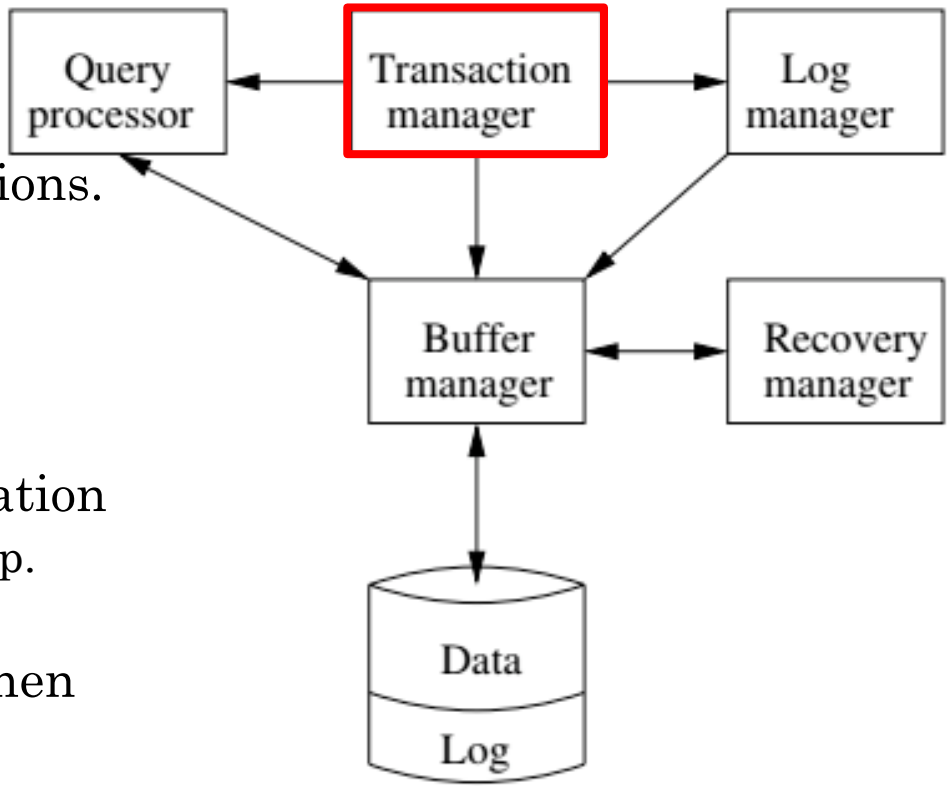
Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism,



Locking

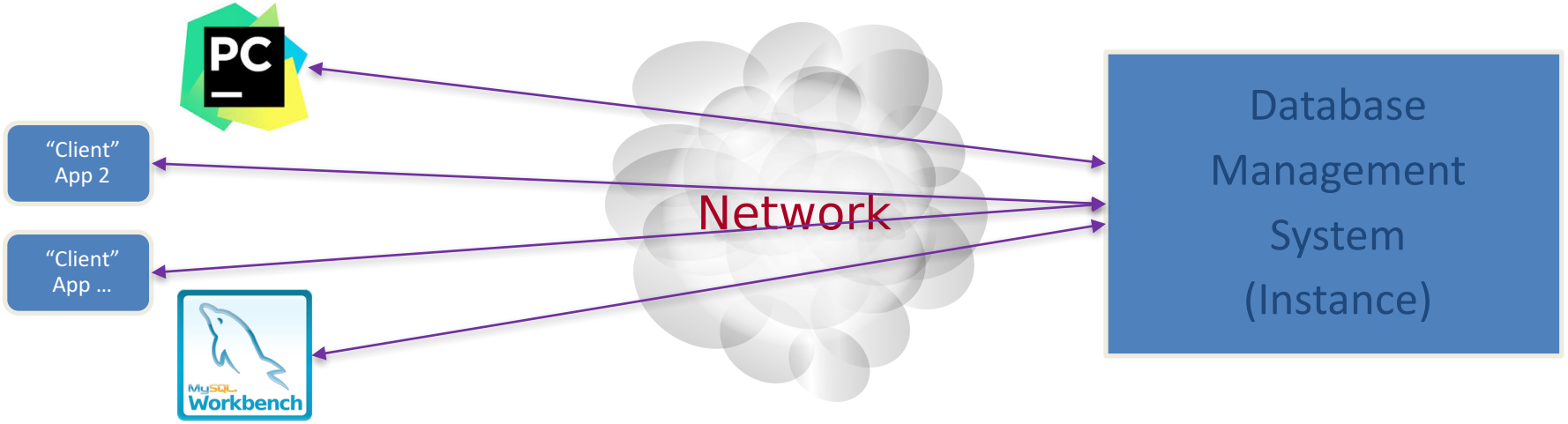
Transaction Manager

- Intercepts all database operations.
- Acquires/checks locks on
  - Records
  - Pages
  - Index pages
- Suspends and queues an operation
  - In another active, executing op.
  - Has a conflicting lock.
- Restarts queued operations when conflicting locks are released.



García-Molina et al., p. 846

Applications and Database Servers



- Applications *connect* to a database (server) engine. An application may have one or more connections.
  - A connection *may* have one or more simultaneous *sessions*.
  - A *session* may have one or more active *cursors*.
  - Applications execute SQL statements on a cursor.
- An application may begin, commit and abort transactions on a *session*.
- An application may set an *isolation level* for a connection or session.

A lot of these actions happen implicitly or via defaults, e.g. autocommit.

You have to take explicit steps to bypass and used some of the functions.

Some Demos

```
from Samples.Transactions import tx_utils
import json

def test_1():
    cnx = tx_utils.get_new_connection()
    tx_utils.set_isolation_level(cnx, "SERIALIZABLE")
    cid = input("Input customer ID: ")
    bid = input("Input new balance:")
    tx_utils.insert(cnx, "accounts",
                    ['customer_id', 'balance'],
                    [cid, bid], commit=False)
    print("Insert performed.")
    decision = input("Enter C for commit and anything else f
    if decision == 'C':
        tx_utils.commit_cnx(cnx)
    else:
        tx_utils.abort_cnx(cnx)

test_1()
```

```
set session transaction isolation level serializable;
-- set session transaction isolation level repeatable read;
select * from accounts;
```

- Two different client programs connecting to the database.
  - MySQL Workbench
  - PyCharm
- Both have
  - Turned ”autocommit” off.
  - Set an isolation.
  - Explicitly commit/abort.
- Show some cases:
  - SQL Workbench blocks PyCharm due to serializability and vice versa.
  - Repeatable read:
    - SQL Workbench sees the same result set despite PyCharm insert.
    - Sees new result after commit and starting a new transaction.

TX_UTILS Code

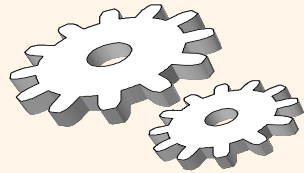
```
import pymysql

pymysql_exceptions = (
    pymysql.err.IntegrityError,
    pymysql.err.MySQLError,
    pymysql.err.ProgrammingError,
    pymysql.err.InternalError,
    pymysql.err.DatabaseError,
    pymysql.err.DataError,
    pymysql.err.InterfaceError,
    pymysql.err.NotSupportedError,
    pymysql.err.OperationalError)

default_dbhost = "localhost"                    # Changeable defaults in constructor
default_port = 3306
default_dbname = "transactions"
default_dbuser = "dbuser"
default_dbpw = "dbuser"
cursorClass = pymysql.cursors.DictCursor        # Default setting for DB connections
charset = 'utf8mb4'

def get_new_connection(autocommit=False):
    cnx = pymysql.connect(
        host=default_dbhost,
        port=default_port,
        user=default_dbuser,
        password=default_dbpw,
        db=default_dbname,
        charset=charset,
        cursorclass=pymysql.cursors.DictCursor,
        autocommit=autocommit)
    return cnx
```

TX_UTILS Code

```
def insert(cnx, table_name, columns, values, commit=True):
    """
    This is a helper method to perform an insert.
    :param table_name: The RDB table for the insert.
    This is a table in the catalog,
    not one of the CSV table names.
    :param columns: Columns names.
    :param values: Matching values.
    :return: Return value from insert statement
    """
    q = "insert into " + table_name + " "
    column_count = len(columns)
    column_list = ",".join(columns)
    column_list = "(" + column_list + ")"
    v = ["%s"] * column_count
    v = ",".join(v)
    v = " values (" + v + ")"
    q += " " + column_list + " " + v
    rr = run_q(cnx, q, values, False, commit=commit)
    row_id = run_q(cnx,
       "SELECT LAST_INSERT_ID() AS inserted_row_id",
           None, True, commit=commit)
    return row_id


def set_isolation_level(cnx, il):
    q = "set session transaction isolation level " + il
    result = run_q(cnx, q, None, commit=False, fetch=False)
```

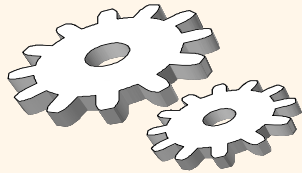
```
def commit_cnx(cnx):
    print("Committing.")
    cnx.commit()
    cnx.close()


def abort_cnx(cnx):
    print("Aborting.")
    cnx.rollback()
    cnx.close()


def run_q(cnx, q, args, fetch=False, commit=True):
    """
    :param cnx: The database connection to use.
    :param q: The query string to run.
    :param args: Parameters to insert into query template if q is a template.
    :param fetch: True if this query produces a result and the function shoul
        perform and return fetchall()
    :param commit: If True, commit the transaction
    :return:
    """
    result = None

    try:
        cursor = cnx.cursor()
        print("\nExecuting statement = ", q, "args=", args)
        result = cursor.execute(q, args)
        if fetch:
            result = cursor.fetchall()
        if commit:
            print("\nCommitting for statement = ", q, "args=", args)
            cnx.commit()
    except pymysql_exceptions as original_e:
        raise(original_e)

    return result
```

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

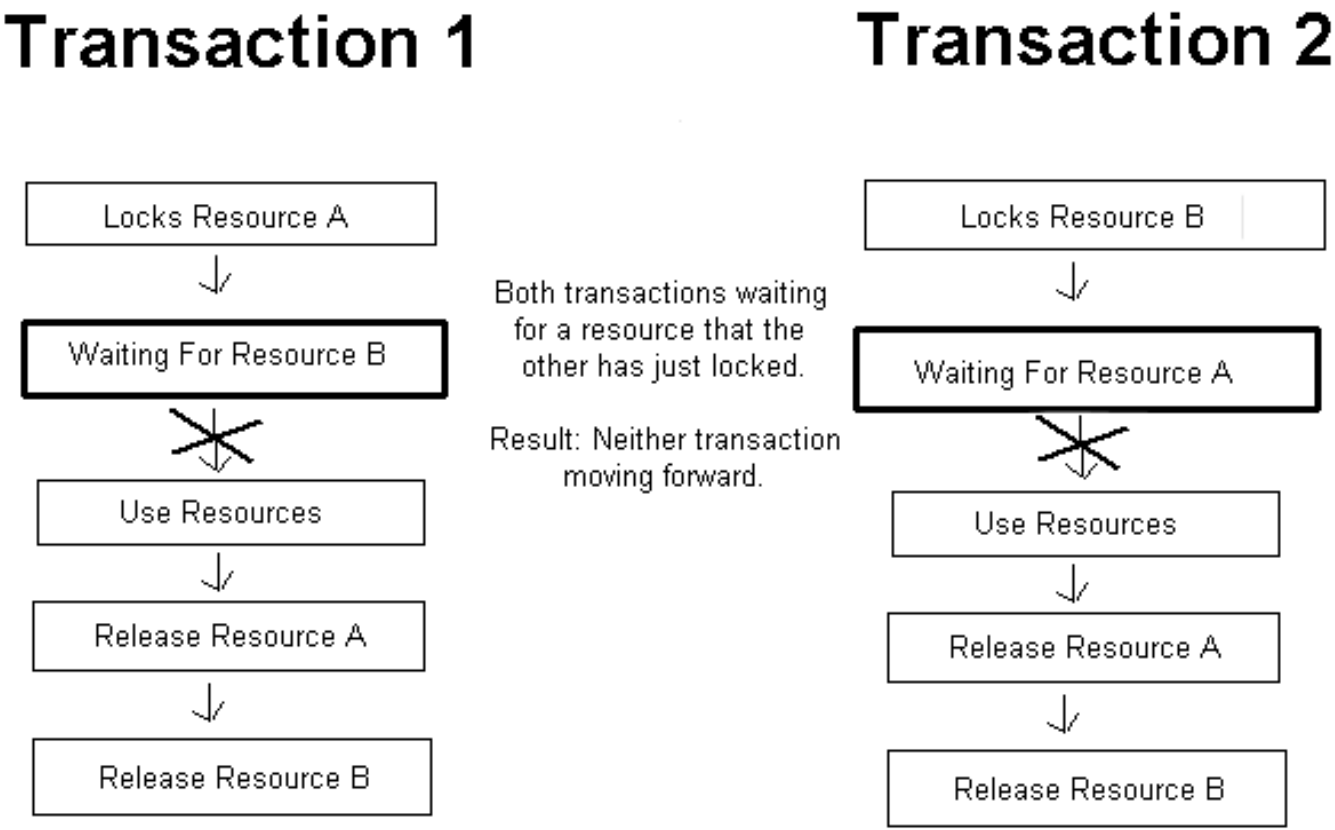
In Databases, Cursors Define *Isolation*

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); }
```

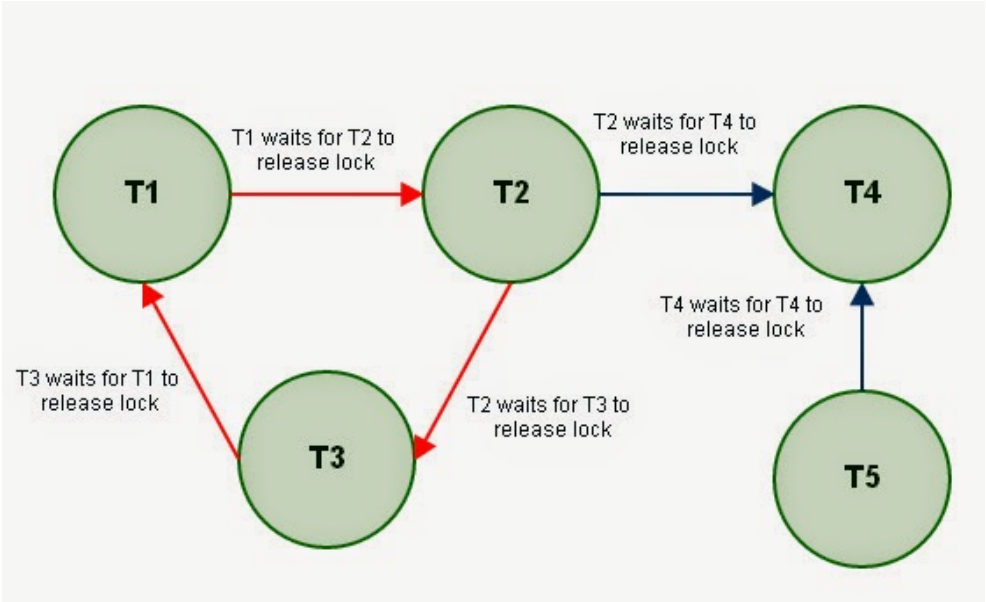
- We have talked about ACID transactions
  - Atomic: A set of writes all occur or none occur.
  - Durable: The data "does not disappear," e.g. write to disk.
  - Consistent: My applications move the database between consistent states, e.g. the transfer function works correctly.
- Isolation
  - Determines 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.*

Deadlock



Waitfor Graph

- The waitfor graph has a directed edge from T1 to T2 if
  - T1 requests a lock on a resource.
  - T2 currently holds an incompatible lock on the resource.
- Delete the edge when T2 releases the conflicting lock.
- There is a *deadlock* if the waitfor graph has a directed cycle.
- Breaking deadlocks:
  - Use the waifor graph and explicitly abort a transaction.
  - The more common approach is to use ”timeouts,” retry the transaction and hope the problem goes away,



Optimistic and Pessimistic Concurrency Control

- Locking is *pessimistic*.
  - It achieves isolation and serializability.
  - But sometimes prevents concurrency when a conflict does not actually happen.
- **"Optimistic concurrency control** (**OCC**) is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. OCC assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.
- OCC is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods."
  (https://en.wikipedia.org/wiki/Optimistic_concurrency_control)

Optimistic Concurrency Control

Under **optimistic concurrency control**, transactions proceed in **three phases**:

1. **Read phase:**

   Execute transaction, but do **not** write data back to disk immediately. Instead, collect updates in the transaction's **private workspace**.

2. **Validation phase:**

   When the transaction wants to **commit**, test whether its execution was correct (only acceptable conflicts happened). If not, **abort** the transaction.

3. **Write phase:**

   Transfer data from private workspace into database.

Note: phases 2 and 3 need to be performed in a non-interruptible critical section (also called **val-write phase**).

http://joerg.endrullis.de/databases/06_transactions.pdf/0224_slide.html

Optimistic Concurrency Control: Validation

Validation is typically implemented by maintaining:

- a **read set RS(T_k)** (attributes read by T_k), and
- a **write set WS(T_k)** (attributes written by T_k)

for every transaction T_k.

Backward-oriented optimistic concurrency control (BOCC)

On commit, compare T_k against all **committed** transactions T_i.
Check **succeeds** if

T_i committed before T_k started **or** $\text{RS}(T_k) \cap \text{WS}(T_i) = \varnothing$

Forward-oriented optimistic concurrency control (FOCC)

On commit, compare T_k against all **running** transactions T_i.
Check **succeeds** if

$$\text{WS}(T_k) \cap \text{RS}(T_i) = \varnothing$$

Transactions and REST
(Remember HW 2)

REST

- "Representational State Transfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. Web Services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by **using a uniform and predefined set of stateless operations.** Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations." (Emphasis added).(https://en.wikipedia.org/wiki/Representational_state_transfer)
-

- The six core characteristics of the REST style are:Client–server architecture
  - Statelessness
  - Cacheability
  - Layered system
  - Code on demand (optional)
  - Uniform interface

State

- REST Statelessness is easy to misunderstand.
- The server *clearly* has long-lived state information, e.g.
  - Account balances.
  - Customer contact information.
  - Product catalog information in a database.
  - etc.
- Client-Server interactions have two types of state:
  - Resource state
  - *Conversation / Session*
- "In computer science, in particular networking, a session is a temporary and interactive information interchange between two or more communicating devices, or between a computer and user." (https://en.wikipedia.org/wiki/Session_(computer_science))

State

- REST Statelessness is easy to misunderstand.
- The server *clearly* has long-lived state information, e.g.

Connections, sessions and cursors are “conversation statefull” and conflict with REST, which makes transactions difficult.

REST applications use a form of optimistic concurrency control, explicit conflict detection and application level recovery.

We saw a similar problem with LIMIT and OFFSET in HW 2

- [illegible]s,

or between a computer and user."
(https://en.wikipedia.org/wiki/Session_(computer_science))

Client – Server State and Cursors Concept

(Logical) Client

```
from Samples.Transactions import stateful_cursor

first = stateful_cursor.get_by_last_name("Williams")

print("First = ", first)

done = False
while not done:
    next = stateful_cursor.get_next()
    if next is None or len(next) == 0:
        done = True
    else:
        print("Next = ", next)
```

(Logical) Server

```
import pymysql.cursors
import pandas as pd
import json


cnx = pymysql.connect(host='localhost',
                      user='dbuser',
                      password='dbuser',
                      db='lahman2017',
                      charset='utf8mb4',
                      cursorclass=pymysql.cursors.DictCursor)

cursor = cnx.cursor()


def get_by_last_name(lastName):
    cursor.execute("select * from people where nameLast=%s",(lastName));
    r = cursor.fetchone()
    return r

def get_next():
    r = cursor.fetchone()
    return r
```

Stateful session.

Client – Server State and Cursors Concept

(Logical) Client

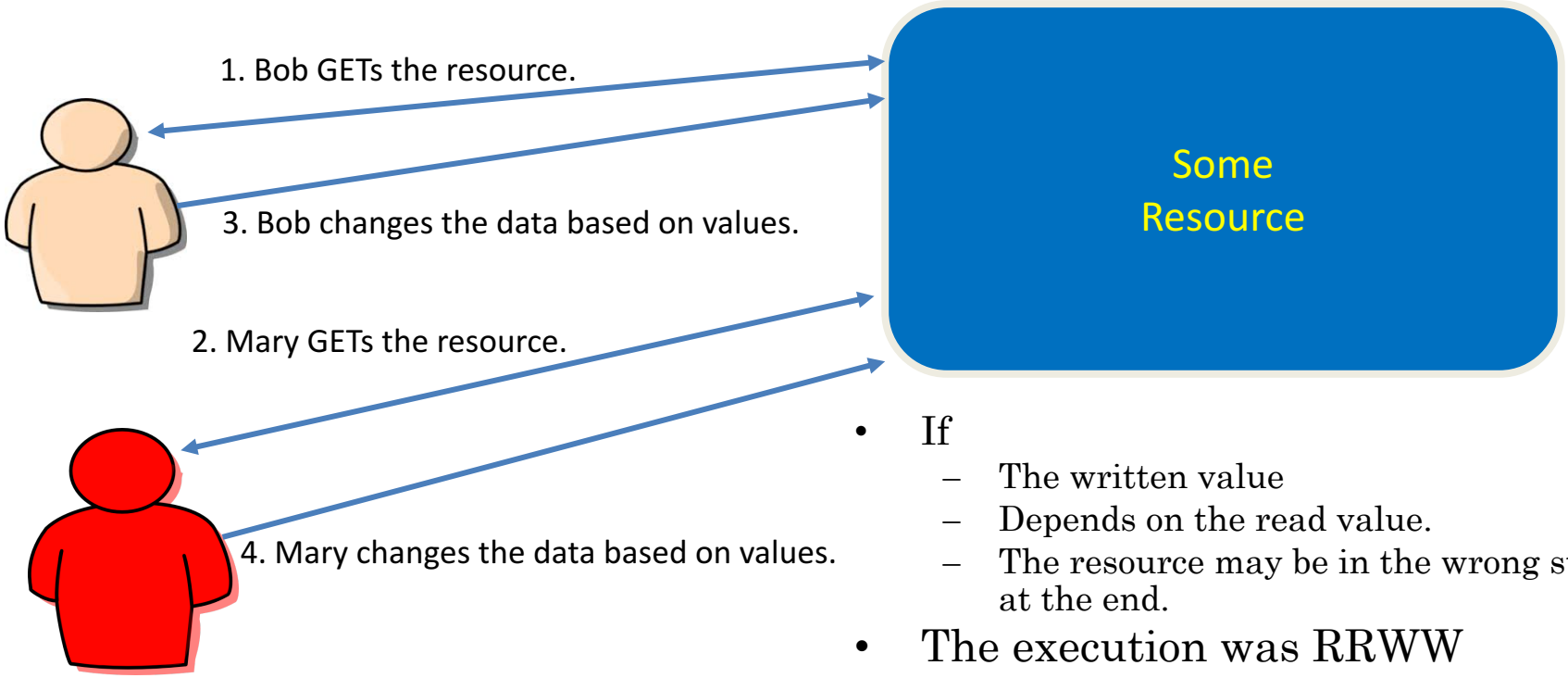
```
from Samples.Transactions import stateless_no_cursor


done = False
offset = 0

while not done:
    next = stateless_no_cursor.get_by_last_name("Williams", offset)
    if next is None or len(next) == 0:
        done = True
    else:
        print("Next = ", next)
        offset += 1
```

(Logical) Server

```
import pymysql.cursors
import pandas as pd
import json


cnx = pymysql.connect(host='localhost',
                      user='dbuser',
                      password='dbuser',
                      db='lahman2017',
                      charset='utf8mb4',
                      cursorclass=pymysql.cursors.DictCursor)


def get_by_last_name(lastName, offset):
    cursor=cnx.cursor()
    q = "select * from people where nameLast=%s limit 1 offset %s"
    cursor.execute(q, (lastName, offset))
    r = cursor.fetchone()
    return r
```

Stateless interaction

Read then Update Conflict

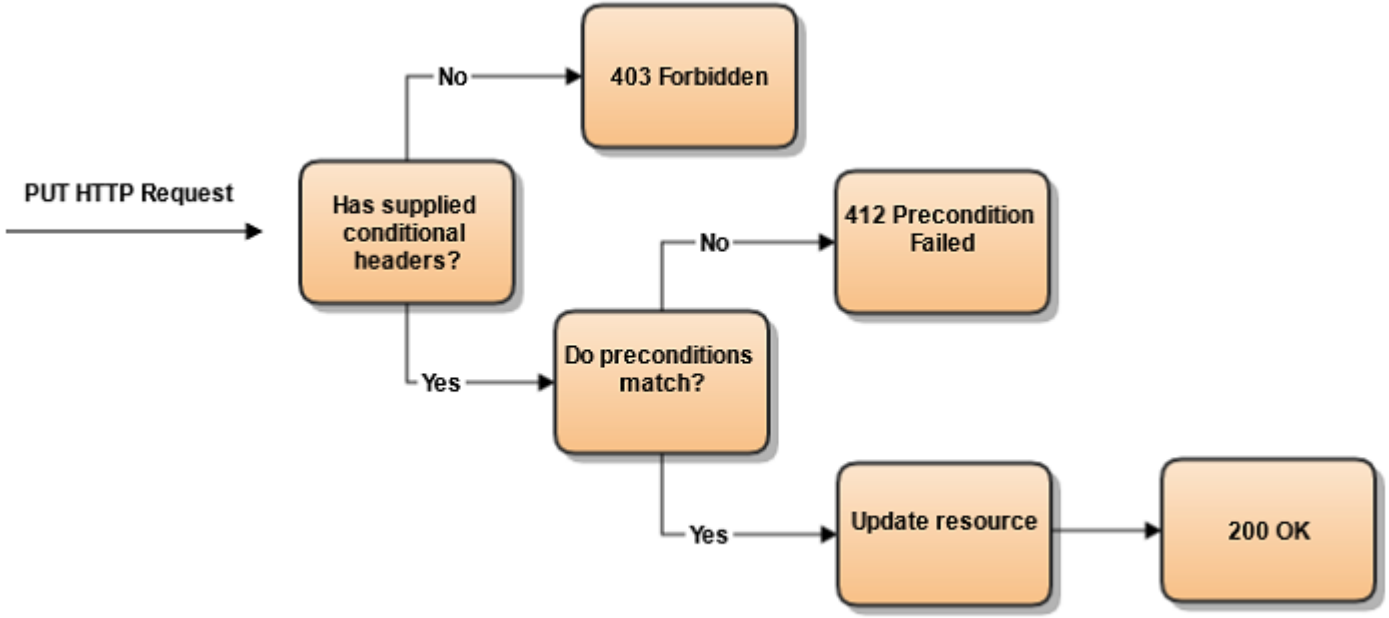


- If
  - The written value
  - Depends on the read value.
  - The resource may be in the wrong state at the end.
- The execution was RRWW
- But the intent was RWRW.

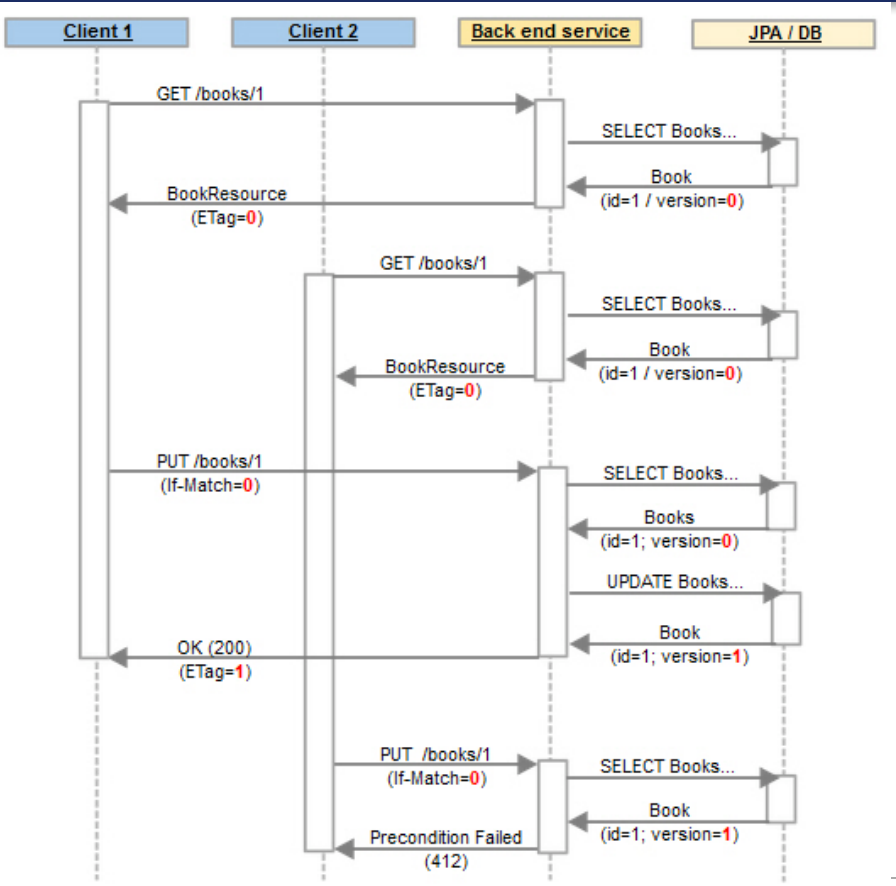
Isolation/Concurrency Control

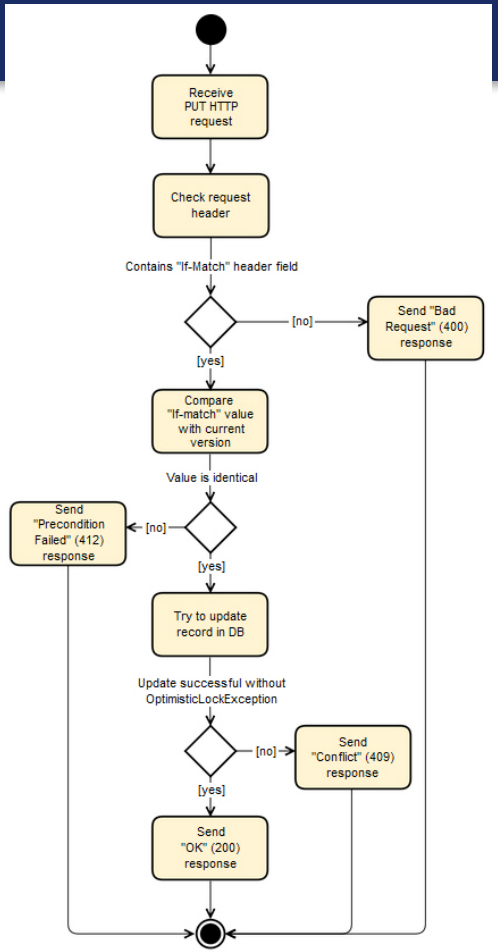
- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

Conditional Processing



ETag Processing





NoSQL Databases

Introduction

Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

A **NoSQL** (originally referring to "non SQL" or "non relational")[1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.[3][4][5] NoSQL databases are increasingly used in big data and real-time web applications.[6] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[7][8]

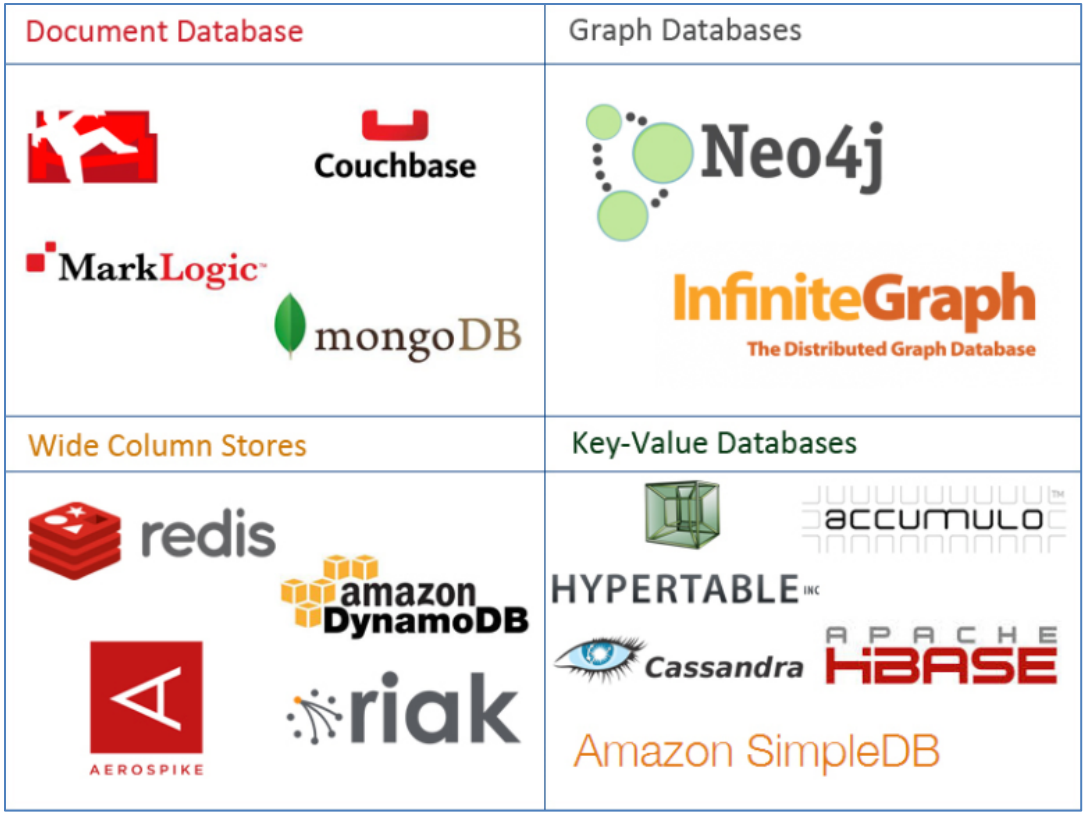
Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.[9]

Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

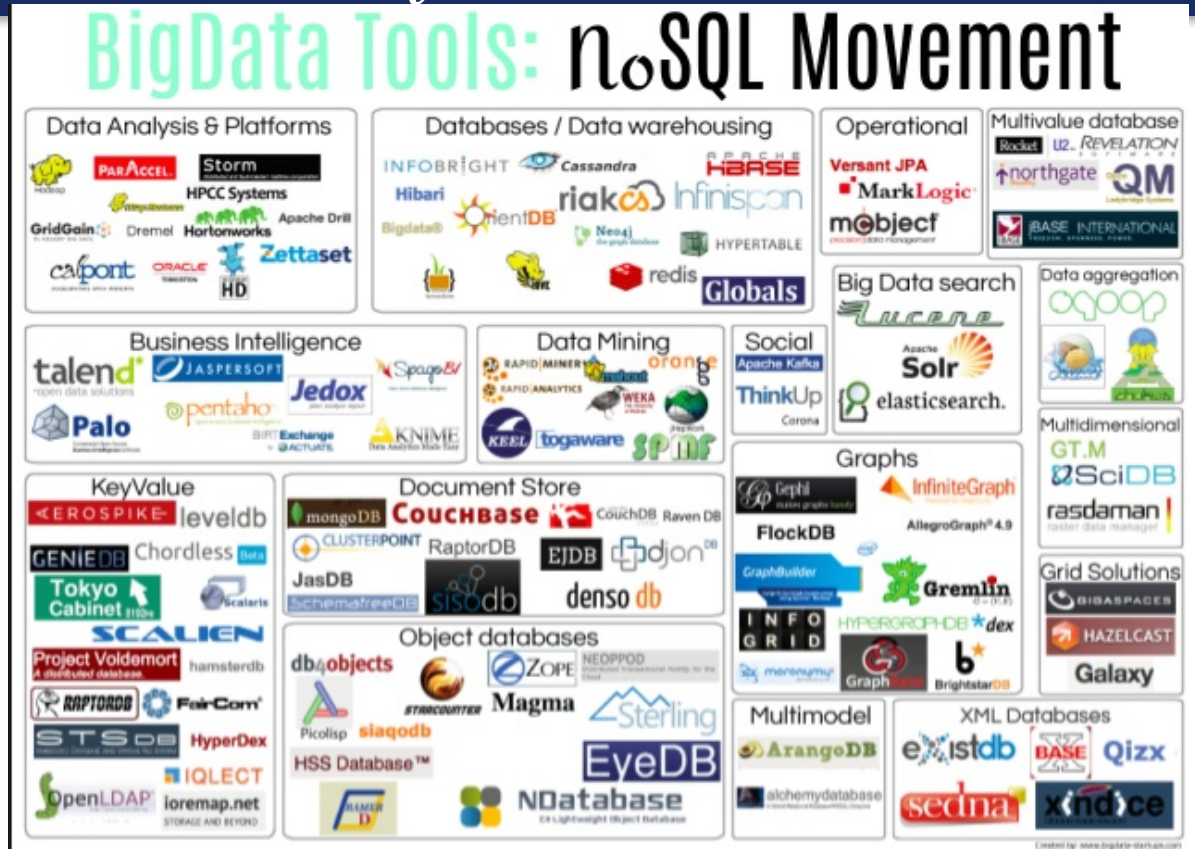
"Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, … …

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of data loss.[12] Fortunately, some NoSQL systems provide concepts such as write-ahead logging to avoid data loss.[13] For distributed transaction processing across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[14]

One Taxonomy



Another Taxonomy



CAP Theorem

ACID

CAP Theorem

- Consistency

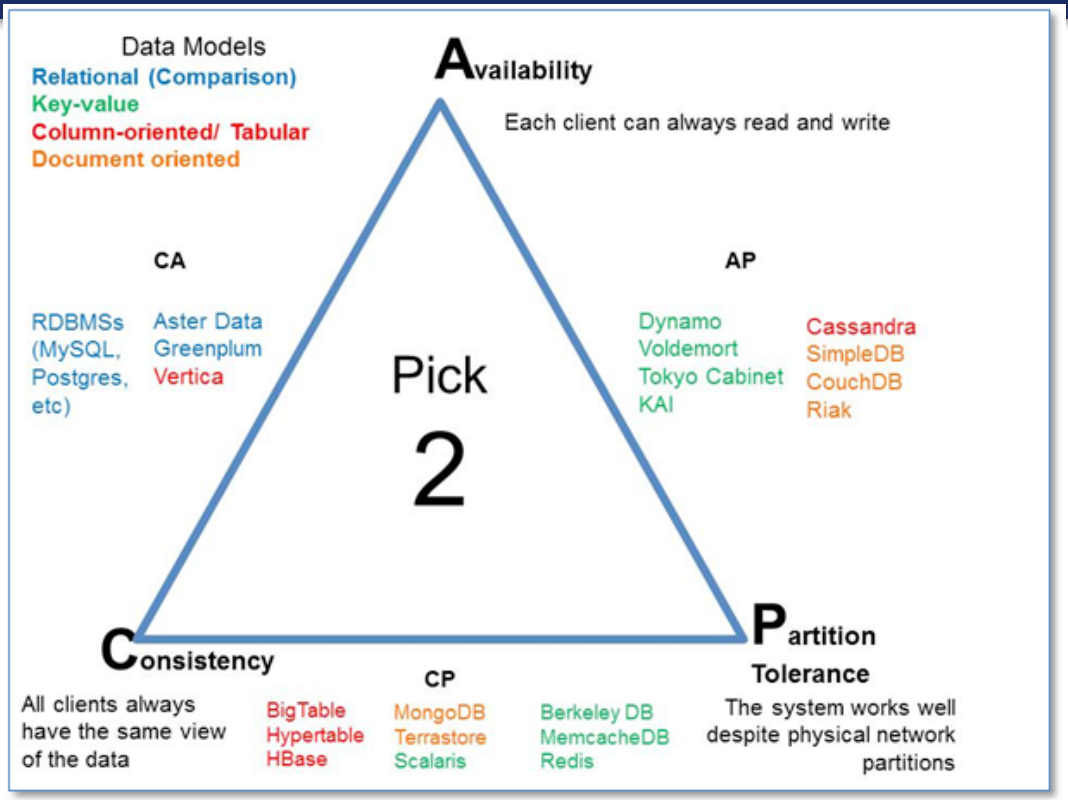
  Every read receives the most recent write or an error.

- Availability

  Every request receives a (non-error) response – without guarantee that it contains the most recent write.

- Partition Tolerance

  The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

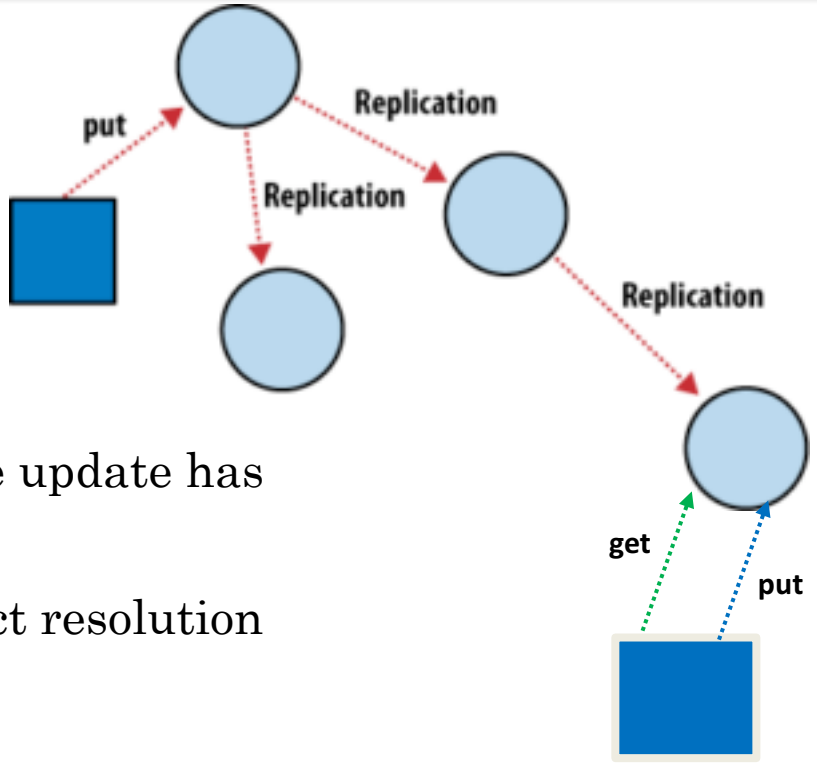


Consistency Models

- **STRONG CONSISTENCY:** Strong consistency is a consistency model where all subsequent accesses to a distributed system will always return the updated value after the update.
- **WEAK CONSISTENCY:** It is a consistency model used in distributed computing where subsequent accesses might not always be returning the updated value. There might be inconsistent responses.
- **EVENTUAL CONSISTENCY:** Eventual consistency is a special type of weak consistency method which informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Eventual Consistency

- Availability and scalability via
  - Multiple, replicated data stores.
  - Read goes to "any" replica.
  - PUT/POST/DELETE
    - Goes to any replica
    - Change propagate asynchronously
- GET may not see the latest value if the update has not propagated to the replica.
- There are several algorithms for conflict resolution
  - Detect and handle in application.
  - Clock/change vectors/version numbers
  - … …

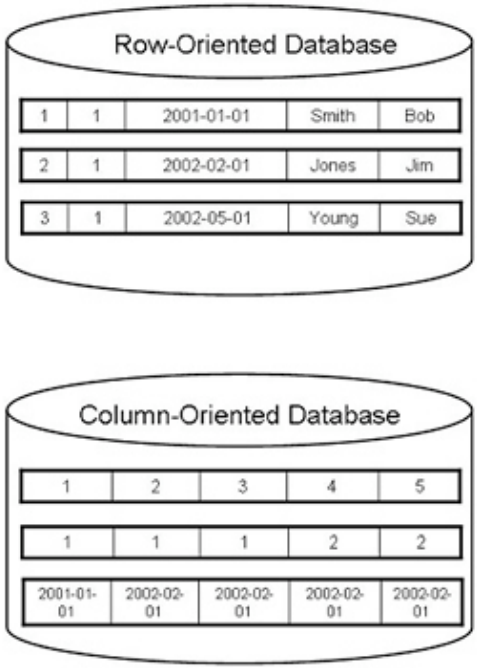


An Aside -- Columnar

Columnar (Relational) Database

- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for different types of query, e.g.
  - Columnar is extremely powerful for BI scenarios
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) to get a few columns
  - Row is powerful for OLTP. Transaction typically create and retrieve
    - One row at a time
    - All the columns of a single row.

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

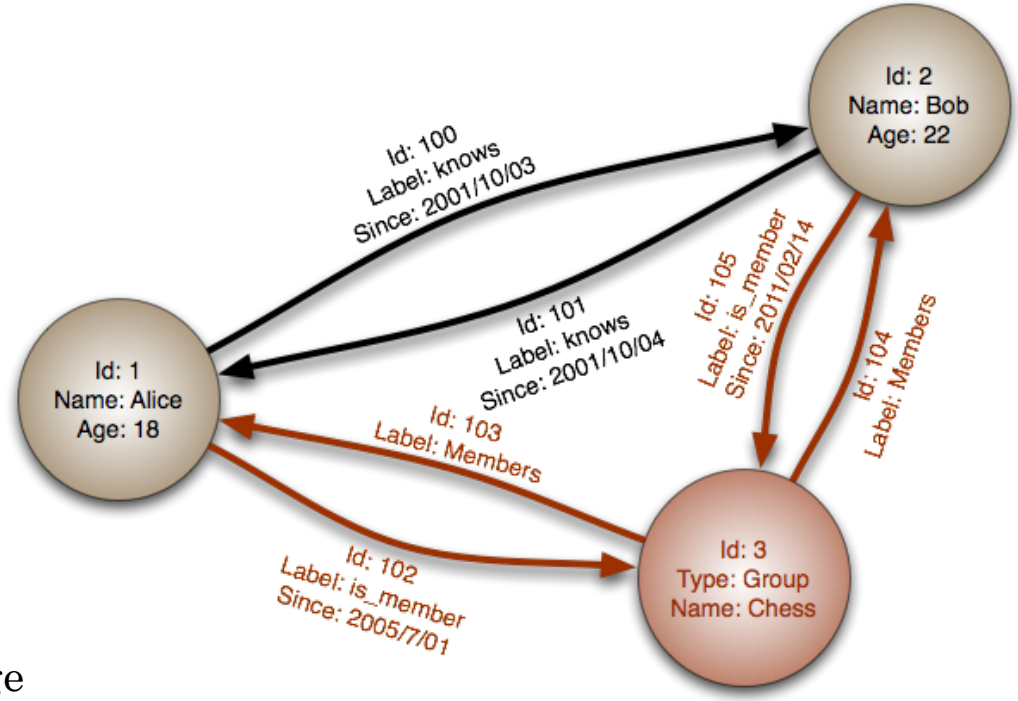


(https://www.dbbest.com/blog/column-oriented-database-technologies/)

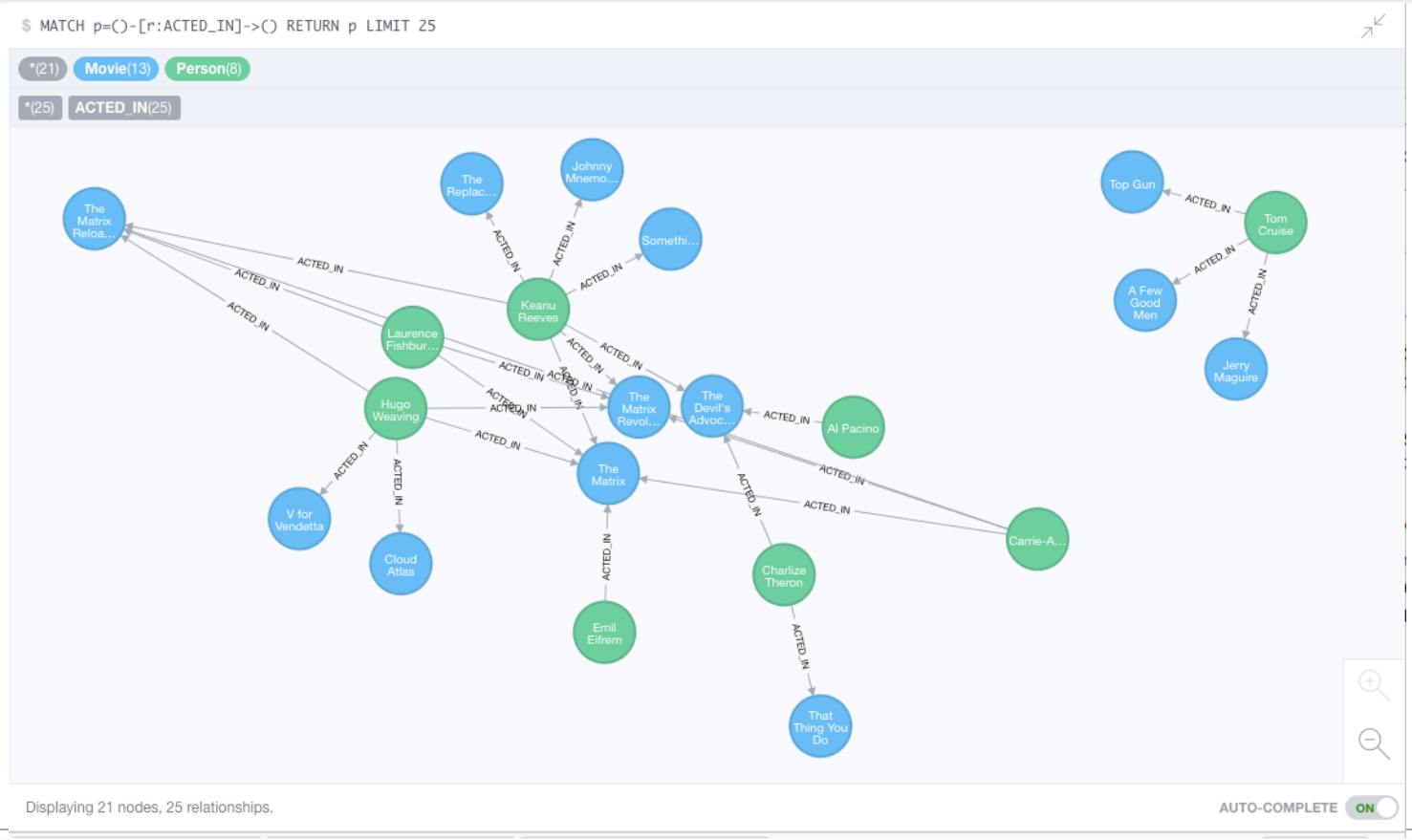
Graph Databases

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

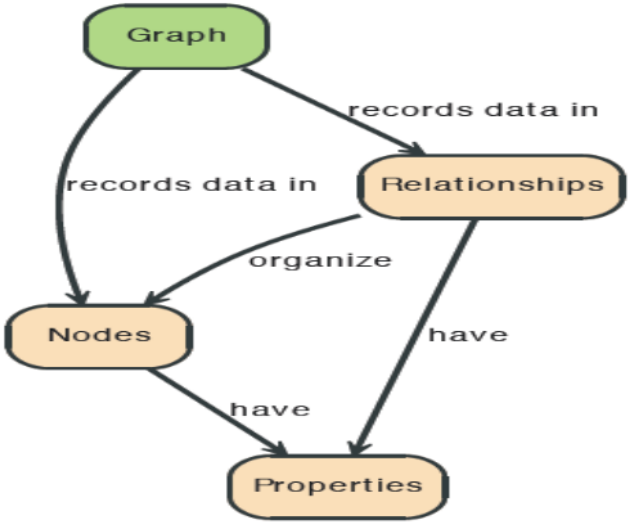
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

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!

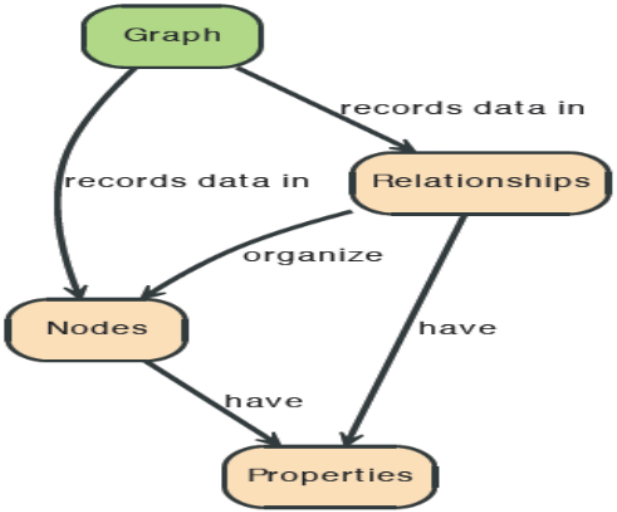
Graphs

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



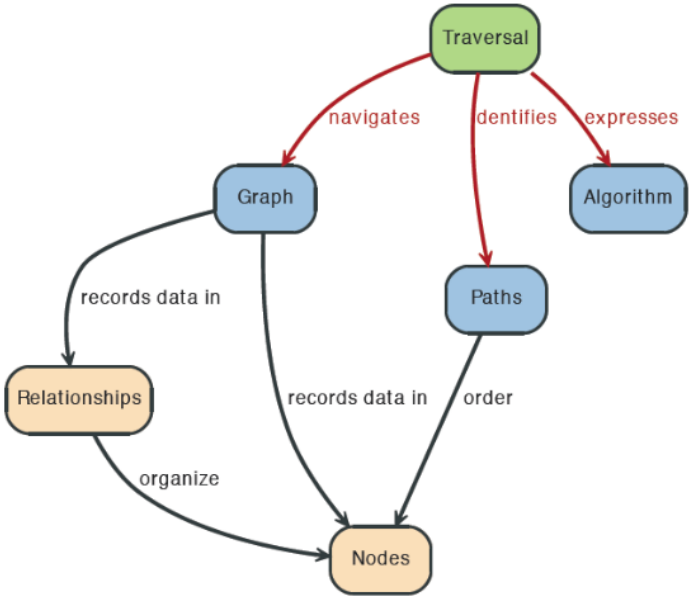
Graphs

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



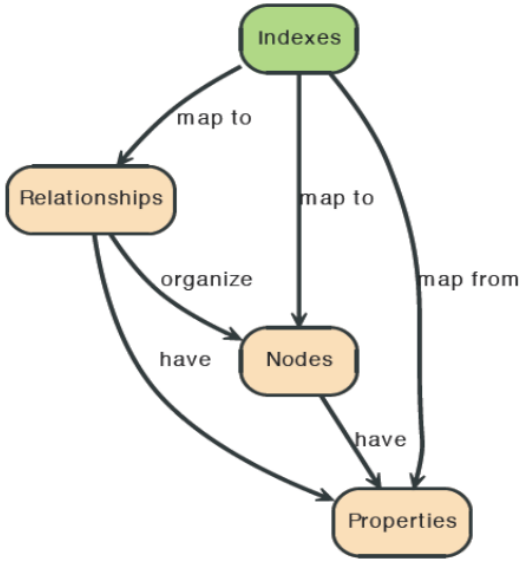
Query a graph with Traversal

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

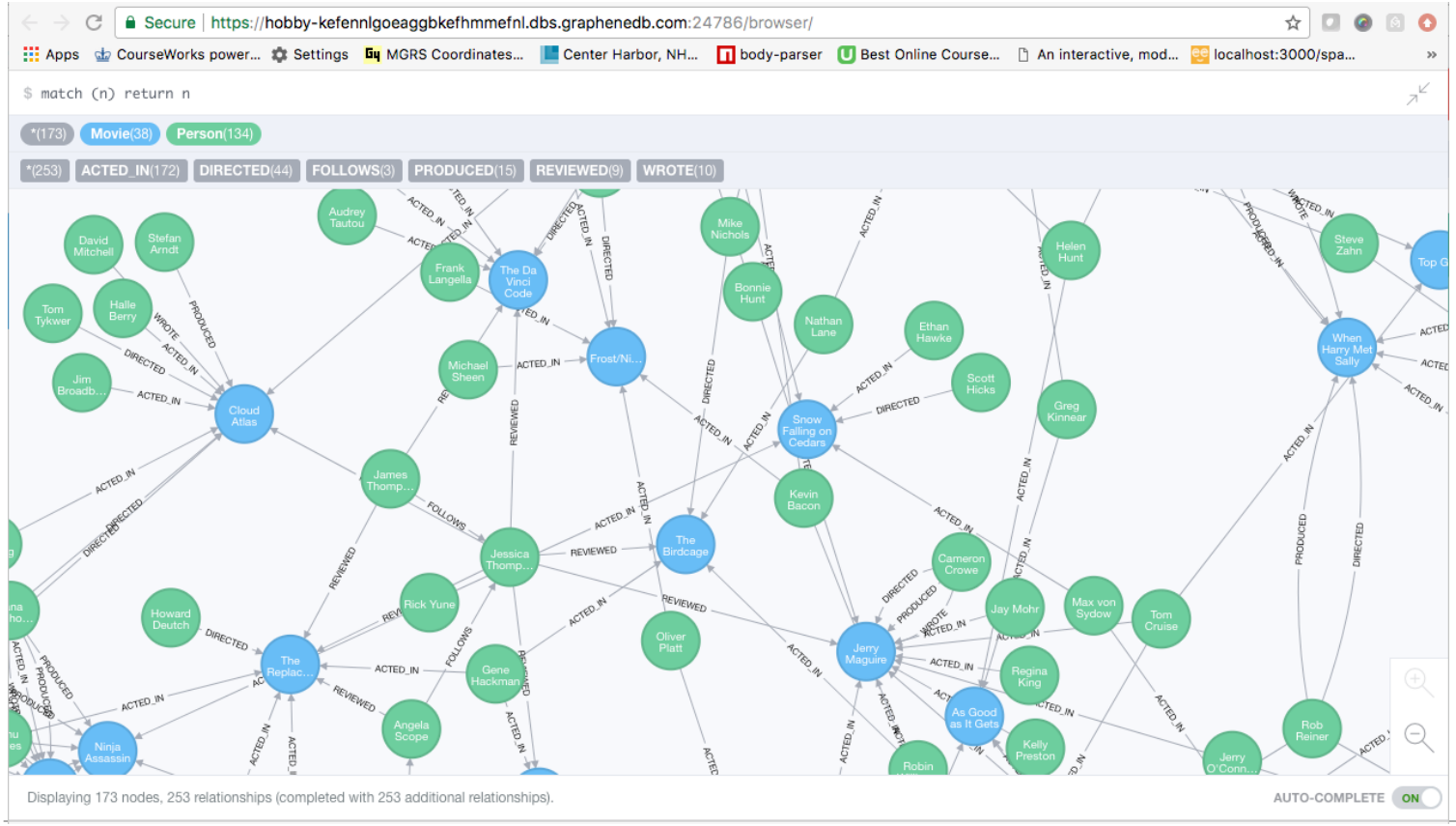


Indexes

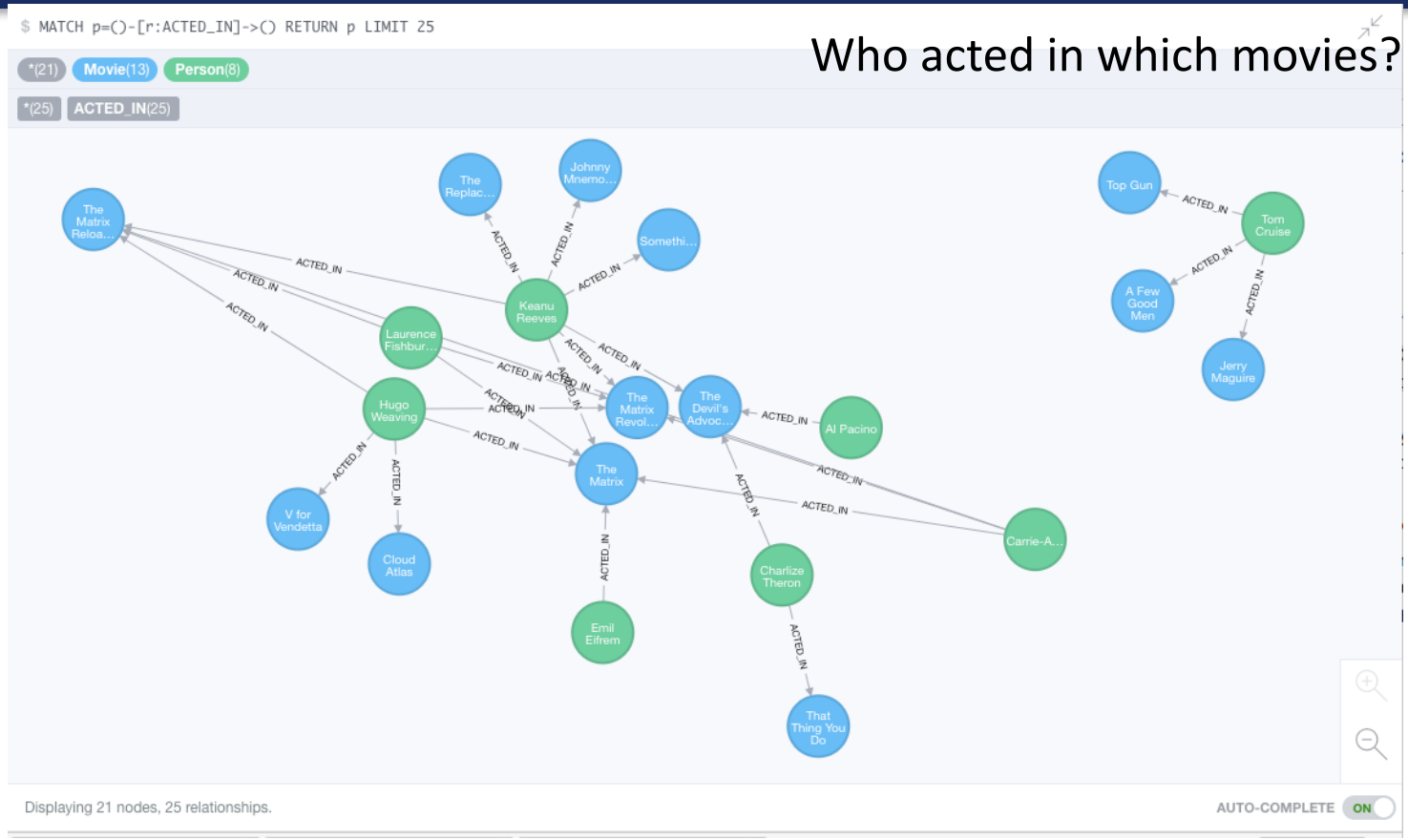
- "An Index —maps from→ Properties —to either→ Nodes or Relationships"



A Graph Database (Sample)



Neo4J Graph Query



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...

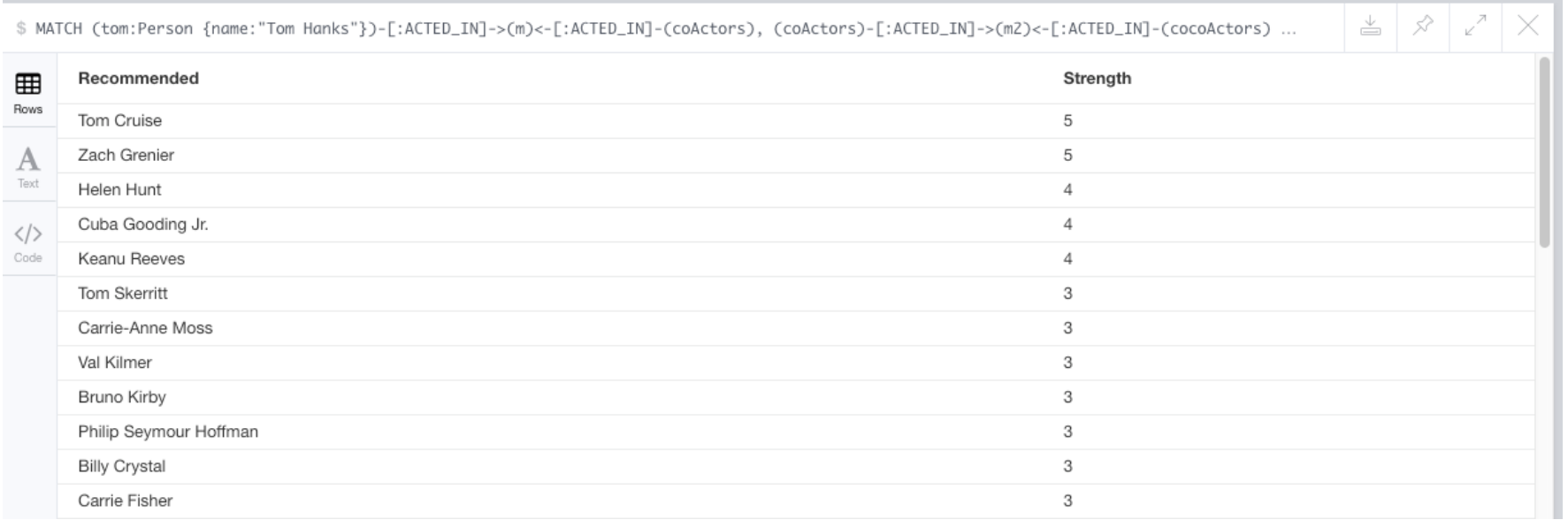
```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

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
```

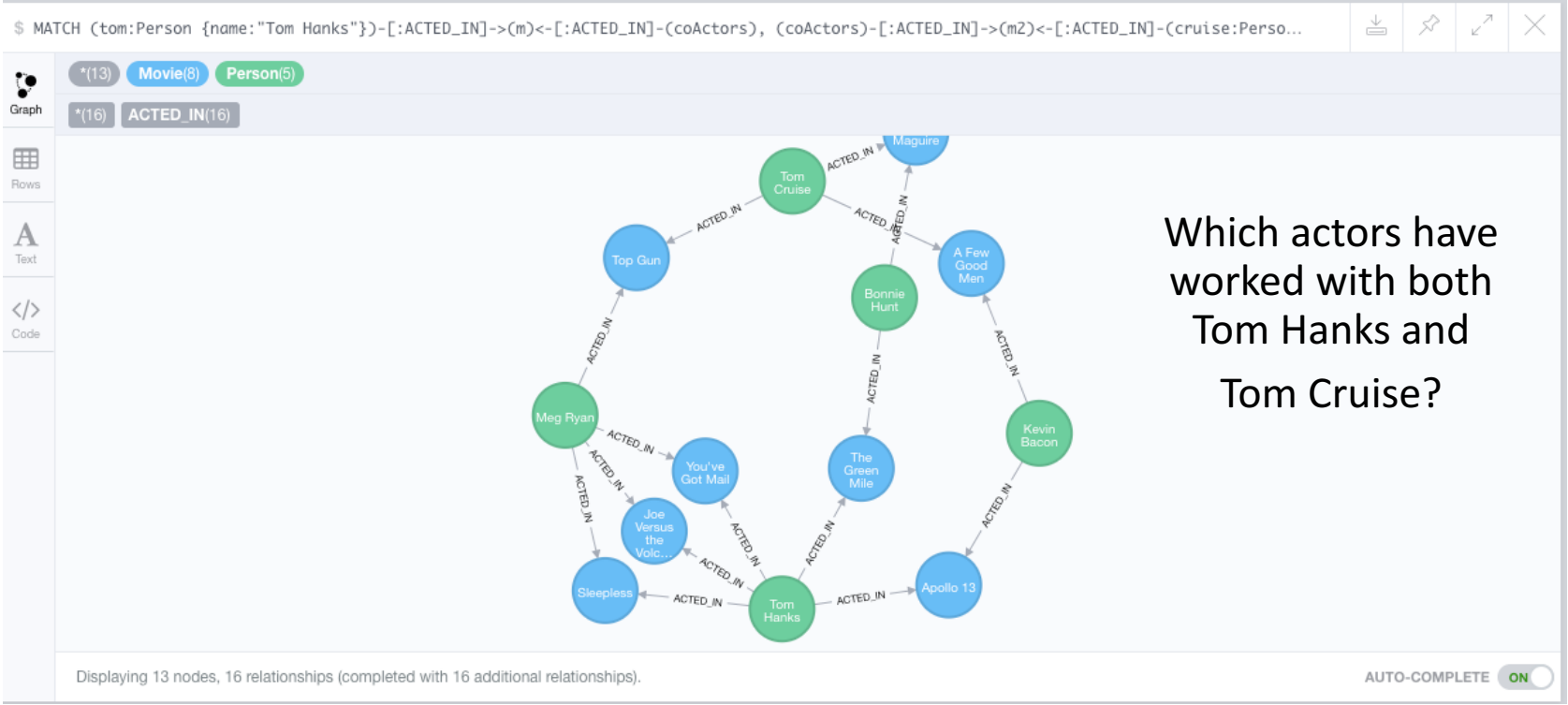
Recommend

```
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
```



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
```



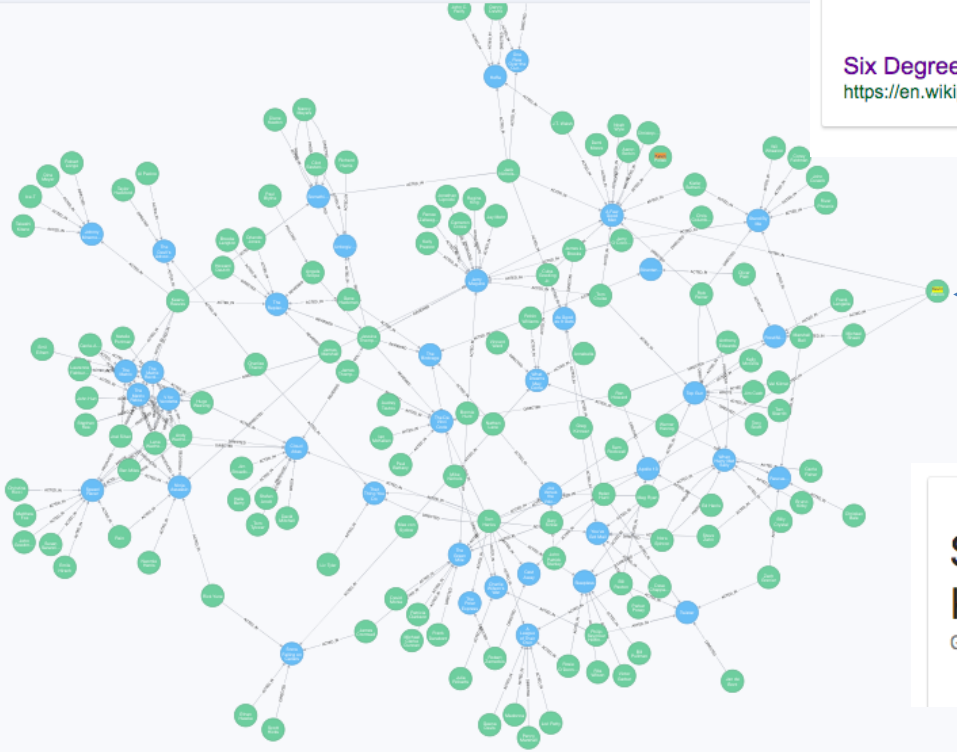
Which actors have worked with both Tom Hanks and

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)

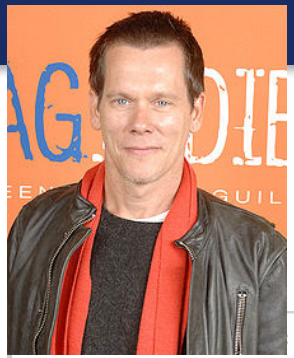


Six **Degrees of Kevin Bacon** is a parlour game based on the "six **degrees** of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin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

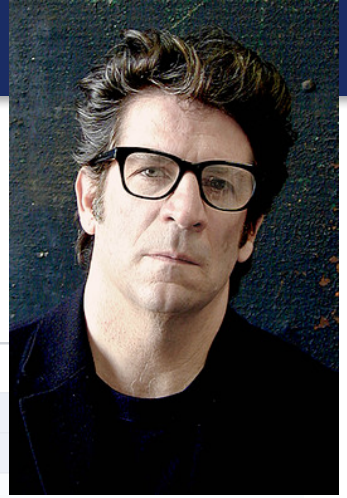
Six Degrees of Kevin Bacon - Wikipedia
https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

About this result Feedback

Six Degrees of Kevin Bacon
Game



How do you get from Kevin Bacon to Robert Longo?



```
'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)