Database Management Systems

Grant and Revoke

Prof. Bhargavi M & Prof. Vidhyashree DM

Dept of Computer Science and Engineering

Introduction to DB Security

- Secrecy: Users should not be able to see things they are not supposed to.
  - E.g., A student can't see other students' grades.

- Integrity: Users should not be able to modify things they are not supposed to.
  - E.g., Only instructors can assign grades.

- Availability: Users should be able to see and modify things they are allowed to.

Threats to Database

- Loss of integrity: Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creating, inserting, and updating data; changing the status of data; and deleting data. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts.
- Loss of availability: Loss of availability occurs when the user or program cannot access these objects.
- Loss of confidentiality: Database confidentiality refers to the protection of data from unauthorized disclosure.

Access Controls

- A security policy specifies who is authorized to do what.

- A security mechanism allows us to enforce a chosen security policy.

Two main mechanisms at the DBMS level:

- Discretionary access control
- Mandatory access control

Discretionary Access Control

- Based on the concept of access rights or privileges for objects (tables and views), and mechanisms for giving users privileges (and revoking privileges).
- Creator of a table or a view automatically gets all privileges on it.
- DBMS keeps track of who subsequently gains and loses privileges, and ensures that only requests from users who have the necessary privileges (at the time the request is issued) are allowed.

Mandatory security mechanisms

These are used to enforce multilevel security by classifying the data and users into various security classes and then implementing the appropriate security policy of the organization.

Database Management Systems

Grant Permission

GRANT privileges ON object TO users [WITH GRANT OPTION]

The following privileges can be specified:

- SELECT: Can read all columns (including those added later via ALTER TABLE command).
- INSERT(col-name): Can insert tuples with non-null or non-default values in this column.
  INSERT means same right with respect to all columns.
- DELETE: Can delete tuples.
- REFERENCES (col-name): Can define foreign keys (in other tables) that refer to this column.

Revoke

Revoke commands are used to revoke the permission granted to users using Grant command

Syntax :

REVOKE [GRANT OPTION FOR] privileges ON object FROM users {RESTRICT | CASCADE}

THANK YOU

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

PES UNIVERSITY ONLINE

Database Management Systems

Introduction to Transaction Processing Concepts and Theory

Prof. Bhargavi M & Prof. Vidhyashree DM

Dept of Computer Science and Engineering

Database Management Systems

Transaction Management and Database Security

PES UNIVERSITY ONLINE

Chapter Outline:

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules Based on Recoverability
- Characterizing Schedules Based on Serializability
- Transaction Support in SQL

Introduction to Transaction Processing Concepts and Theory

Introduction

- Transaction
  - Describes a logical unit of database processing
- Transaction processing systems
  - Systems with large databases and hundreds of concurrent users
  - Require high availability and fast response time

Single-User versus Multiuser Systems

- One criterion for classifying a database system is according to the number of users who can use the system concurrently.
- A DBMS is single-user if at most one user at a time can use the system.
- A DBMS is multiuser if many users can use the system and access the database—concurrently.

Single-User versus Multiuser Systems

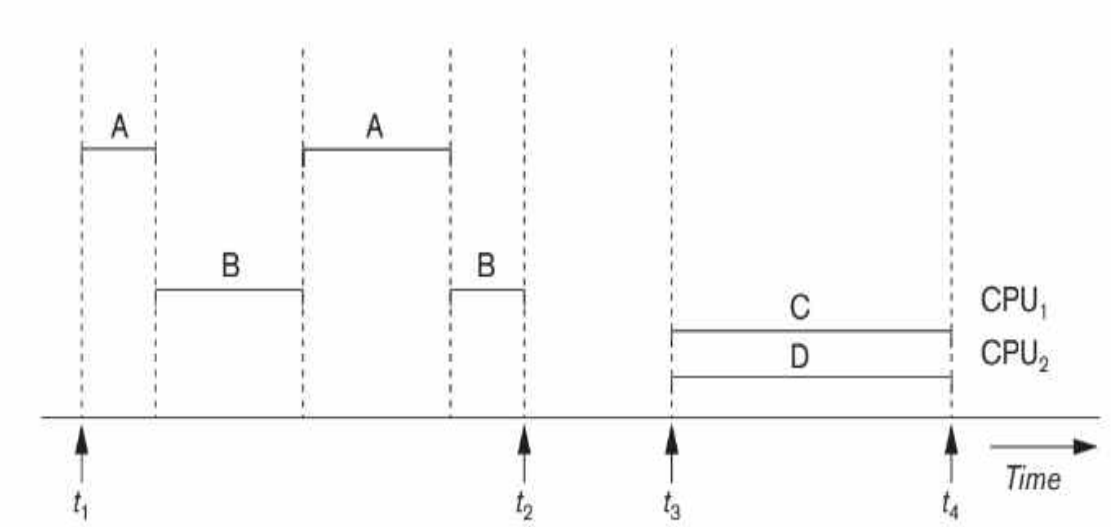


Fig 20.1 Interleaved processing versus parallel processing of concurrent transactions

Introduction to Transaction Processing Concepts and Theory

Transactions

- Transaction: an executing program
  - Forms logical unit of database processing
- Begin and end transaction statements
  - Specify transaction boundaries
- Read-only transaction
- Read-write transaction

Introduction to Transaction Processing Concepts and Theory

Read and Write Operations

read_item(X)

- Reads a database item named X into a program variable named X
- Process includes finding the address of the disk block, and copying it into a buffer in the main memory.Copy item X from the buffer to the program variable named X

- write_item(X)
  - Writes the value of program variable X into the database item named X
- Process includes finding the address of the disk block, copying it into a buffer in the main memory.Copy item X from the program variable named X into its correct location in the buffer, and storing the updated disk block back to disk

Read and Write Operations (cont'd.)

- Read set of a transaction
  - Set of all items read
- Write set of a transaction
  - Set of all items written

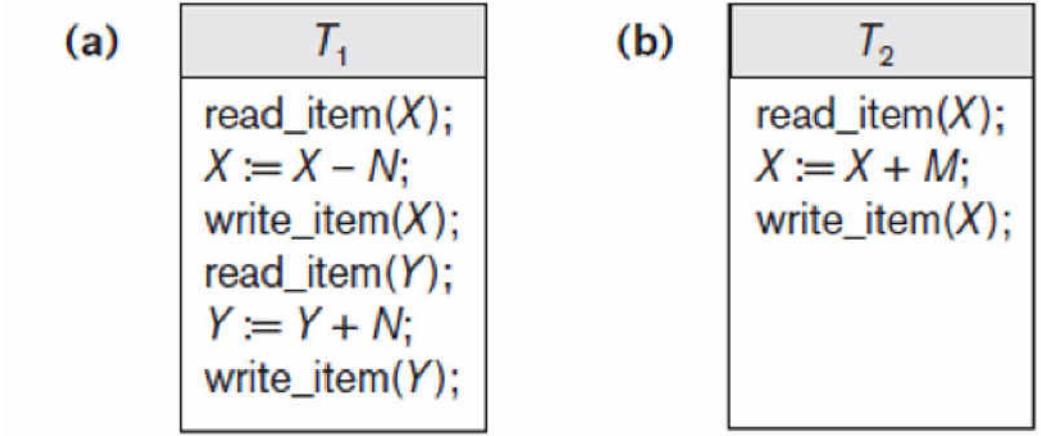


Figure 20.2 Two sample transactions (a) Transaction *T*1 (b) Transaction *T*2

Introduction to Transaction Processing Concepts and Theory

DBMS Buffers

- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a buffer replacement policy is used to choose which buffer will be replaced
  - Example policy: least recently used

Introduction to Transaction Processing Concepts and Theory

Concurrency Control:

- Transactions submitted by various users may execute concurrently
  - Access and update the same database items
  - Some form of concurrency control is needed
- The lost update problem:
  - Occurs when two transactions that access the same database items have operations interleaved
  - Results in incorrect value of some database items

Introduction to Transaction Processing Concepts and Theory

The Lost Update Problem:

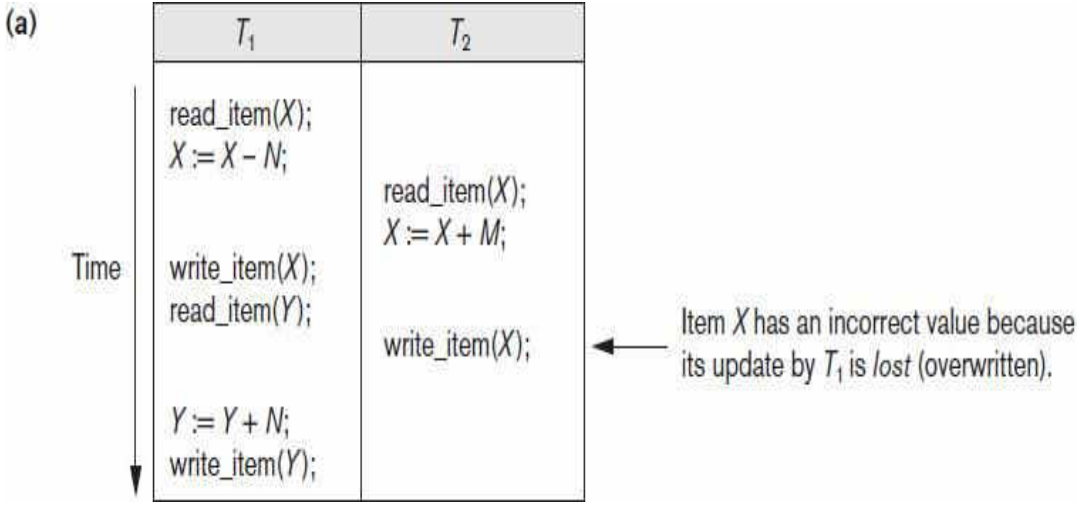
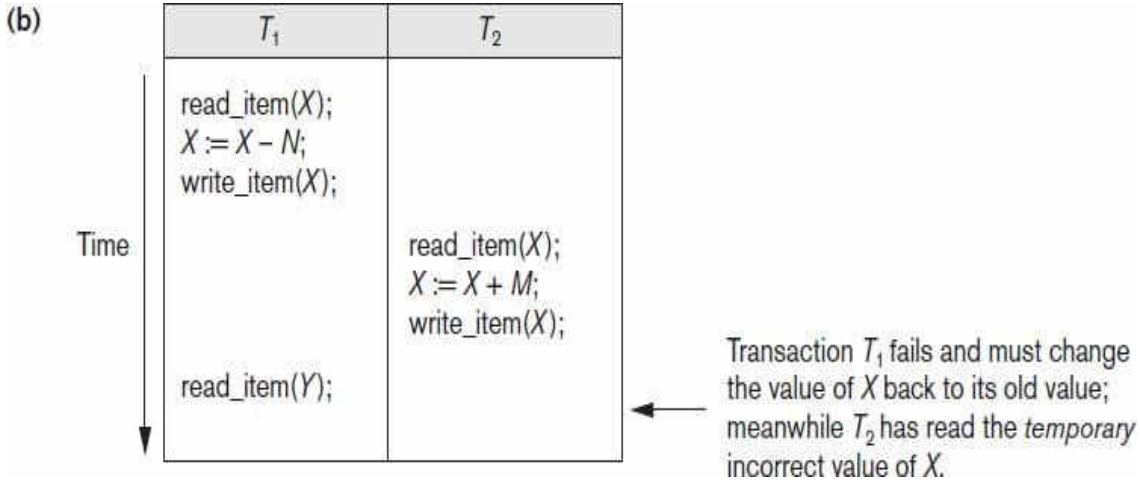


Figure 20.3 Some problems that occur when concurrent execution is uncontrolled (a) The lost update problem

Introduction to Transaction Processing Concepts and Theory

The Temporary Update(or Dirty Read) Problem:

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason



Introduction to Transaction Processing Concepts and Theory

The Incorrect Summary Problem:

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

(c)

T_1	T_3

	sum := 0;
	read_item(A);
	sum := sum + A;
	. . .

read_item(X);
X := X − N;
write_item(X);	

	read_item(X);
	sum := sum + X;
	read_item(Y);
	sum := sum + Y;

read_item(Y);
Y := Y + N;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (c) The incorrect summary problem

Introduction to Transaction Processing Concepts and Theory

The Unrepeatable Read Problem

A transaction T reads the same item twice and the item is changed by another transaction T′ between the two reads. Hence, T receives different values for its two reads of the same item

20.2 Transaction and System Concepts

- System must keep track of when each transaction starts, terminates, commits, and/or aborts
  - BEGIN_TRANSACTION
  - READ or WRITE
  - END_TRANSACTION
  - COMMIT_TRANSACTION
  - ROLLBACK (or ABORT)

Introduction to Transaction Processing Concepts and Theory

Transaction and System Concepts (cont'd.)

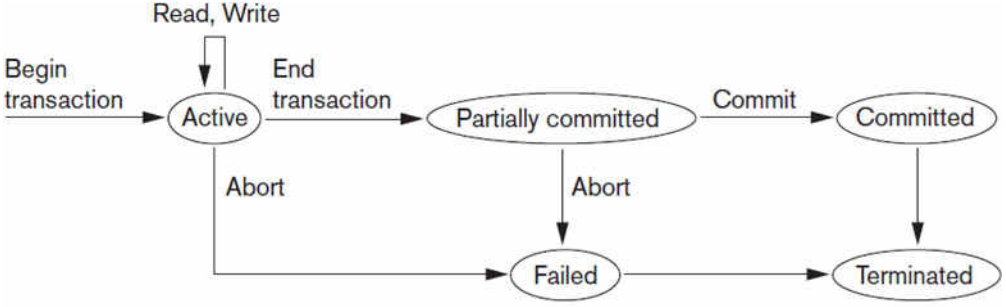


Figure 20.4 State transition diagram illustrating the states for transaction execution

Introduction to Transaction Processing Concepts and Theory

The System Log

- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically
- Undo and redo operations based on log possible

Introduction to Transaction Processing Concepts and Theory

Log Records

T -> Transaction id

1. [start_transaction, T]. Indicates that transaction T has started execution.
2. [write_item, T, X, old_value, new_value]. Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item, T, X]. Indicates that transaction T has read the value of database item X.
4. [commit, T]. Indicates that transaction T has completed successfully and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort, T]. Indicates that transaction T has been aborted.

Introduction to Transaction Processing Concepts and Theory

20.3 Desirable Properties of Transactions

- ACID properties
  - Atomicity
    - Transaction performed in its entirety or not at all
  - Consistency preservation
    - Takes database from one consistent state to another
  - Isolation
    - Not interfered with by other transactions
  - Durability or permanency
    - Changes must persist in the database

20.4 Characterizing Schedules Based on Recoverability

- Schedule or history
  - Order of execution of operations from all transactions
  - Operations from different transactions can be interleaved in the schedule
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Recoverability (cont'd.)

- Two conflicting operations in a schedule
  - Operations belong to different transactions
  - Operations access the same item X
  - At least one of the operations is a write_item(X)
- Two operations conflict if changing their order results in a different outcome
- Read-write conflict
- Write-write conflict

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Recoverability (cont'd.)

- "Recoverable schedules"
  No committed transaction ever needs to be rolled back.
  - Recovery is possible
- Nonrecoverable schedules should not be permitted by the DBMS
- Uncommitted transaction may need to be rolled back
- Cascading rollback may occur in some recoverable schedules

PES UNIVERSITY ONLINE

Characterizing Schedules Based on Recoverability (cont'd.)

- Cascadeless schedule
  - Reads only items that were written by committed transactions.
  - Avoids cascading rollback
- Strict schedule
  - Transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted
  - Simpler recovery process
    - Restore the before image(old value)D

20.5 Characterizing Schedules Based on Serializability

- Serializable schedules
  - Always considered to be correct when concurrent transactions are executing
  - Places simultaneous transactions in series
    - Transaction T_1 before T_2, or vice versa

Introduction to Transaction Processing Concepts and Theory

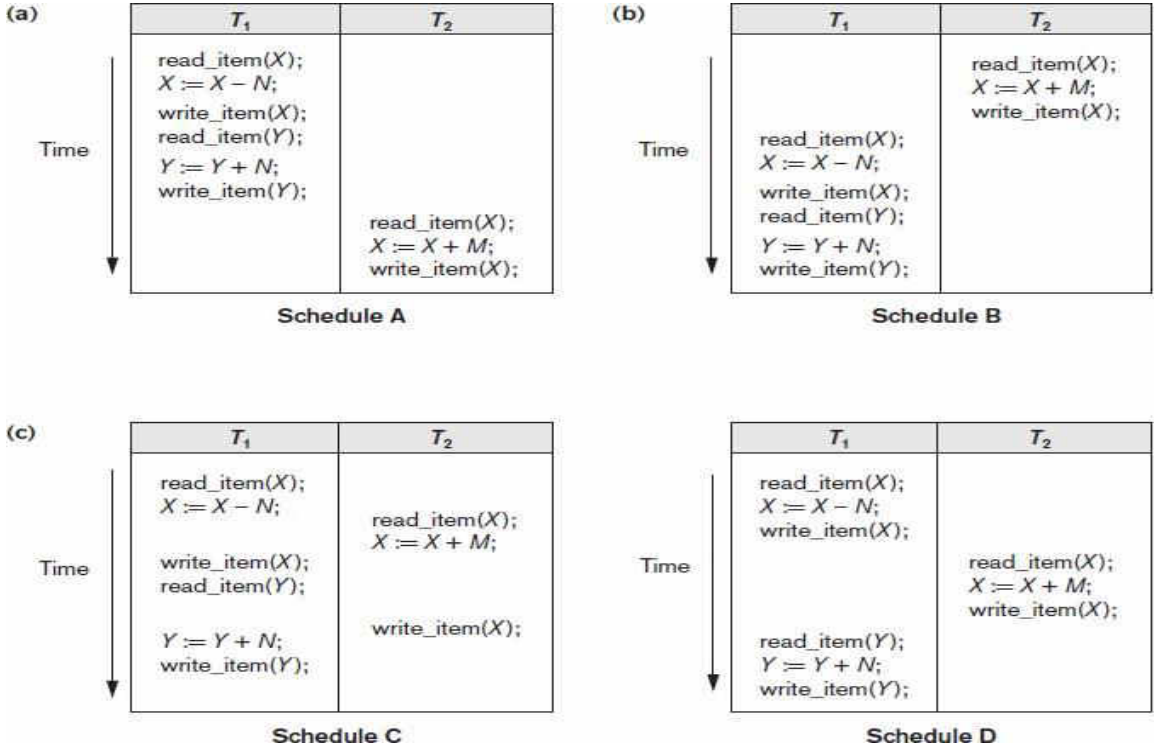


Figure 20.5 Examples of serial and nonserial schedules involving transactions $T1$ and $T2$ (a) Serial schedule A: $T1$ followed by $T2$ (b) Serial schedule B: $T2$ followed by $T1$ (c) Two nonserial schedules C and D with interleaving of operations

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Serializability (cont'd.)

- Problem with serial schedules
  - Limit concurrency by prohibiting interleaving of operations
  - Unacceptable in practice
  - Solution: determine which schedules are equivalent to a serial schedule and allow those to occur
- Serializable schedule of *n* transactions
  - Equivalent to some serial schedule of same *n* transactions

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Serializability (cont'd.)

- Result equivalent schedules
  - Produce the same final state of the database
    - May be accidental
  - Cannot be used alone to define equivalence of schedules

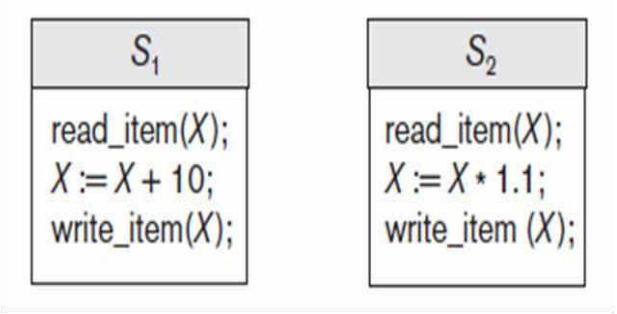


Figure 20.6 Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Serializability (cont'd.)

- Conflict equivalence of two schedules.
  - Relative order of any two conflicting operations is the same in bothschedules
  - If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent.
- Serializable schedules
  - Schedule S is serializable if it is conflict equivalent to some serial schedule S'.

Introduction to Transaction Processing Concepts and Theory

Characterizing Schedules Based on Serializability (cont'd.)

Testing for serializability of a schedule:

- 1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
- 2. For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.
- 3. For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create an edge (Ti → Tj) in the precedence graph.
- 4. For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.
- 5. The schedule S is serializable if and only if the precedence graph has no cycles.

Algorithm 20.1 Testing conflict serializability of a schedule S

Database Management Systems

Introduction to Transaction Processing Concepts and Theory

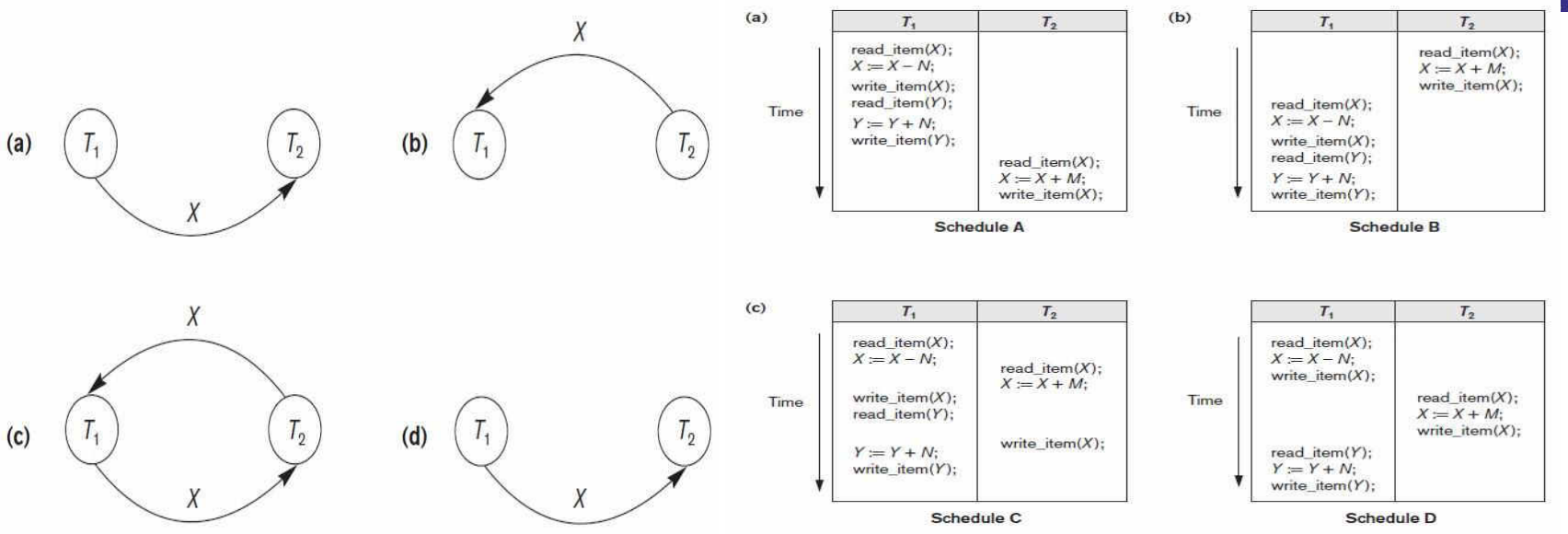


Figure 20.7 Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability (a) Precedence graph for serial schedule A (b) Precedence graph for serial schedule B (c) Precedence graph for schedule C (not serializable) (d) Precedence graph for schedule D (serializable, equivalent to schedule A)

Introduction to Transaction Processing Concepts and Theory

How Serializability is Used for Concurrency Control

- Being serializable is different from being serial
- Serializable schedule gives benefit of concurrent execution
  - Without giving up any correctness
- Difficult to test for serializability in practice
  - Factors such as system load, time of transaction submission, and process priority affect ordering of operations
- DBMS enforces protocols
  - Set of rules to ensure serializability

PES UNIVERSITY ONLINE

View Equivalence and View Serializability

- View equivalence of two schedules
  - As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results
  - Read operations are hence said to see the same view in both schedules.
- View serializable schedule
  - View equivalent to a serial schedule

Introduction to Transaction Processing Concepts and Theory

20.6 Transaction Support in SQL

- No explicit Begin_Transaction statement
- Every transaction must have an explicit end statement
  - COMMIT
  - ROLLBACK
- Access mode is READ ONLY or READ WRITE
- Diagnostic area size option
  - Diagnostic size n, Integer value indicating number of conditions held simultaneously in the diagnostic area.(Conditions-> feedback information on the n most recently executed SQL statement.)

THANK YOU

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

PES UNIVERSITY ONLINE

Database Management Systems

Commit , Rollback and Savepoint

Prof. Bhargavi M & Prof. Vidhyashree DM

Dept of Computer Science and Engineering

SQL Transactions

- SQL Transaction Control Language (TCL) commands are used to manage database transaction.
- SQL transaction command use with DML statement for INSERT, UPDATE and DELETE.
- DML statement are store into SQL buffer until you execute Transaction commands.
- Once you execute transaction commands its store permanent to a database.

Begin and End

Begin statement will indicate start of the transaction and end transaction will indicate end of transaction

Syntax :

```
begin
statements
end transaction;
```

SQL COMMIT

SQL COMMIT command save new changes store into database.

- Syntax :

```
Commit ;
```

SAVEPOINT

- SQL SAVEPOINT command create new save point. SAVEPOINT command save the current point with the unique name in the processing of a transaction.
- Syntax:

```
SAVEPOINT savepoint_name;
```

SQL ROLLBACK

- SQL ROLLBACK command execute at the end of current transaction and undo/undone any changes made since the begin transaction.
- Syntax :

  ROLLBACK [To SAVEPOINT_NAME];

THANK YOU

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

Database Management Systems

Introduction to protocols for Concurrency Control in Databases

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

Database Concurrency Control

- Purpose of Concurrency Control
- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example:

In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

1. **Locking** is an operation which secures
   (a) permission to Read
   (b) permission to Write a data item for a transaction.
   Example:
   Lock (X). Data item X is locked in behalf of the requesting Transaction.

2. **Unlocking** is an operation which removes these permissions from the data item.
   Example:
   Unlock (X): Data item X is made available to all other transactions.

Lock and Unlock are Atomic operations.

Two- Phase Locking system

Binary Locking

- A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity)
- A distinct lock is associated with each database item X.
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
- If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1

- Database requires that all transactions should be well-formed.

A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

Two-Phase Locking Techniques: Essential components

- **Lock Manager**

  Managing locks on data items.

- **Lock Table**

  - Lock manager uses it to store the identity of a transaction locking a data item, the data item, lock mode and pointer to the next data item locked.
  - One simple way to implement a lock table is through linked list.

Two-Phase Locking Techniques: Essential components

lock_item(X):

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is
   unlocked*)  then LOCK (X) ←
   1 (*lock the item*)  else
      begin
       wait (until lock (X) = 0
       And the lock manager wakes up the
   transaction);  goto B
   end;
```

Two-Phase Locking Techniques: Essential components

unlock_item(X):

The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*) if any transactions are waiting
then wake up one of the waiting transactions;

Two-Phase Locking Techniques: Essential components

- **Two locks modes:**
- **Shared /Exclusive Locks (READ/WRITE LOCKS):**
  (a) shared (read)
  (b) exclusive (write).
- **Shared Mode: Read lock (X)**

  More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- **Exclusive Mode: Write lock (X)**

  Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

- UNLOCK MODE.

Lock Table

- keep track of the number of transactions that hold a shared (read) lock on an item in the lock table,
- as well as a list of transaction ids that hold a shared lock.

Each record in the lock table will have four fields:
<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>.

- The system needs to maintain lock records only for locked items in the lock table. (The value (state) of LOCK is either read-locked or write-locked, suitably coded.)
- If LOCK(X) = write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X.
- If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X.

Two-Phase Locking Techniques: Essential components

Lock Conversion

- **Lock upgrade**: existing read lock to write lock
  if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$)
  then convert read-lock (X) to write-lock (X)
  else force Ti to wait until Tj unlocks X
- **Lock downgrade**: existing write lock to read lock
  Ti has a write-lock (X) (*no transaction can have any lock on X*)
  convert write-lock (X) to read-lock (X)

Two-Phase Locking Techniques: The algorithm Two Phases:

(a) Locking (Growing)
(b) Unlocking (Shrinking).

Locking (Growing) Phase:

A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase:

A transaction unlocks its locked data items one at a time.

Requirement:

For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Other variations of 2 phase locking:

Basic:

Transaction locks data items incrementally. This may cause deadlock which is dealt with.

Conservative:

Prevents deadlock by locking all desired data items before transaction begins execution.

Strict 2PL:

A transaction T does not release any of its exclusive (write) locks until after it commits or aborts.

Rigourous 2PL:

A transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

THANK YOU

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

Database Management Systems

Deadlock Detection and Prevention

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

Dealing with Deadlock and Starvation

CONCURRENCY CONTROL SUBSYSTEM

- **Responsible for generating the read_lock and write_lock request**

For example, suppose the system is to enforce the strict 2PL protocol.
Then, whenever transaction T issues a read_item(X), the system calls the read_lock(X) operation on behalf of T.
If the state of LOCK(X) is write_locked by some other transaction T′, the system places T in the waiting queue for item X;
otherwise, it grants the read_lock(X) request and permits the read_item(X) operation of T to execute.

- After each action, the system must update its lock table appropriately.
- Locking is generally considered to have a high overhead, because every read or write operation is preceded by a system locking request.
- The use of locks can also cause two additional problems: deadlock and starvation

Dealing with Deadlock and Starvation

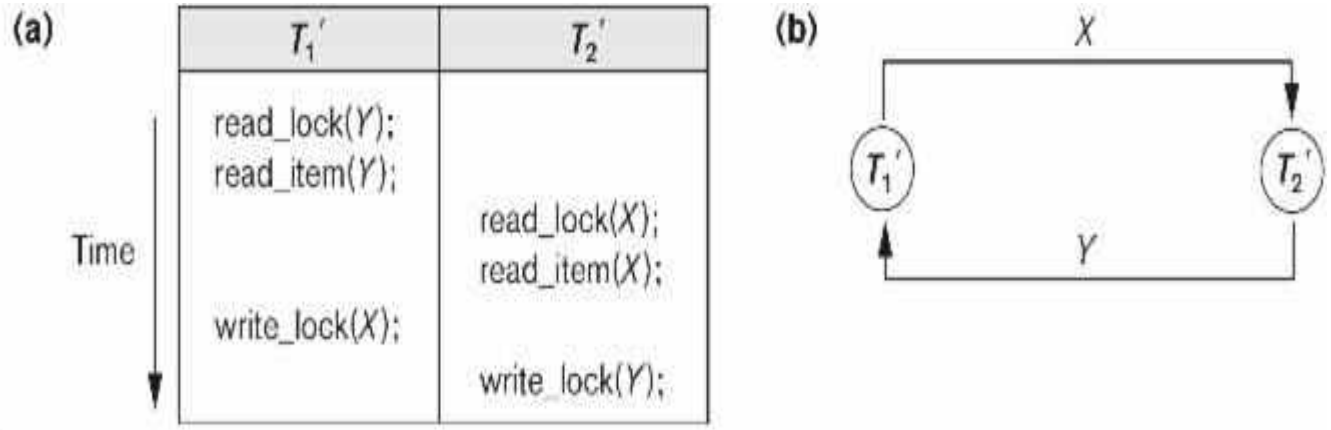


Figure 21.5

Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Dealing with Deadlock and Starvation

Deadlock prevention:

- A transaction locks all data items it refers to before it begins execution.
- This way of locking prevents deadlock since a transaction never waits for a data item.
- The conservative two-phase locking uses this approach.
- limits concurrency

Transaction timestamp TS(T′), which is a unique identifier assigned to each transaction.

The timestamps are typically based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2).

Notice that the older transaction (which starts first) has the smaller timestamp value.

Dealing with Deadlock and Starvation

Two schemes that prevent deadlock are

- wait-die
- wound-wait.

Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by
some other transaction Tj with a conflicting lock.

The rules followed by these schemes are:

- Wait-die. If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to wait; otherwise (Ti younger than Tj) abort Ti (Ti dies) and restart it later with the different timestamp.

- Wound-wait. If TS(Ti) < TS(Tj), then (Ti older than Tj) abort Tj (Ti wounds Tj) and restart it later with the same timestamp; otherwise (Ti younger than Tj) Ti is allowed to wait.

Dealing with Deadlock and Starvation

Deadlock detection and resolution:

In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.

Timeouts : Another simple scheme to deal with deadlock is the use of timeouts. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists.

Dealing with Deadlock and Starvation

Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Dealing with Deadlock and Starvation

Timestamp

- A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
- Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

- **Transaction T issues a write_item(X) operation:**

  If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.

  If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).

- **Transaction T issues a read_item(X) operation:**

  If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.

  If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Timestamp based concurrency control algorithm

- Strict Timestamp Ordering

  Transaction T issues a write_item(X) operation:

  If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

  Transaction T issues a read_item(X) operation:

  If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Timestamp based concurrency control algorithm

Thomas's Write Rule:

A modification of the basic TO algorithm, known as
Thomas's write rule, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:
If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.

If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.

If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

THANK YOU

Prof. Bhargavi M & Prof. Vidhyashree DM

Dept. of Computer Science and Engineering

Database Management Systems

Performance, Query Execution Plan, Indexing

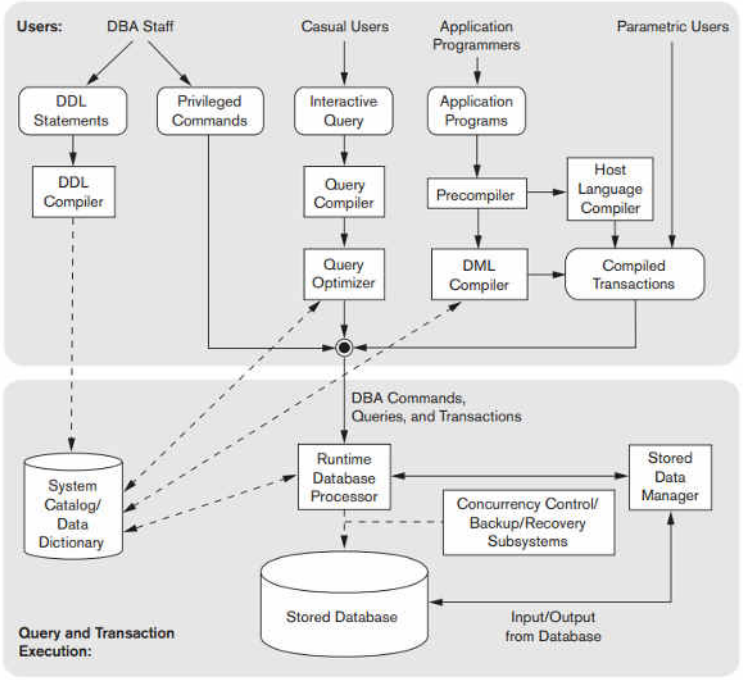
Suresh Jamadagni

Department of Computer Science and Engineering

Database Management Systems

Query Optimizer

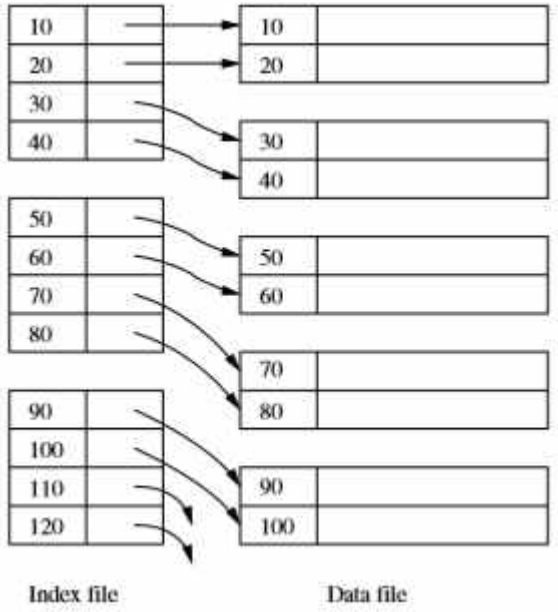
- The query optimizer examines alternative table and index access paths, operator algorithms, join orderings, join methods, parallel execution distribution methods, and so on.
- It chooses the execution plan with the lowest estimated cost.
- The optimizer calculates this cost based on the estimated usage of resources (such as I/O and CPU time), and memory needed to execute the query.
- The estimated query cost is a relative number proportional to the expected elapsed time needed to execute the query with the given execution plan
- The goal of cost based optimization is to find the best trade-off between the lowest run time and the least resource utilization



Database Management Systems

Indexes

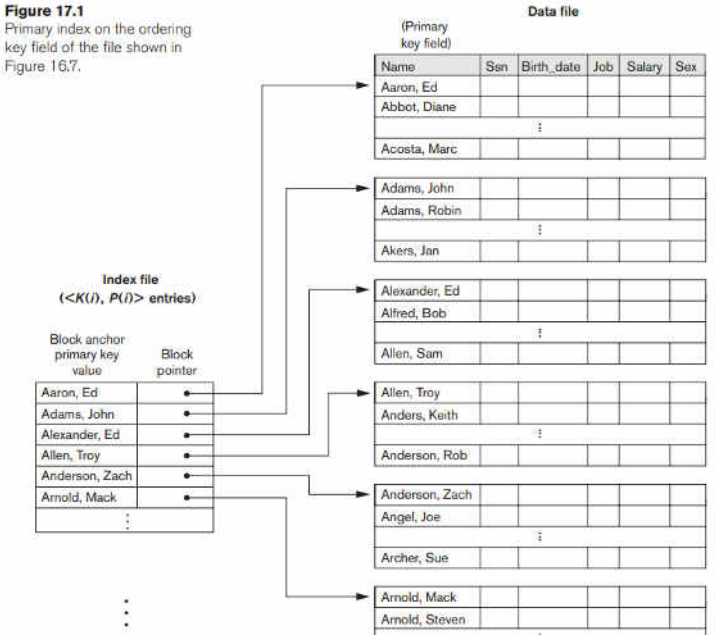
- Index is an auxiliary structure used for faster retrieval of records based on search criteria
- A data file is used to store tuples of a table.
- The data file may have one or more index files.
- Each index file associates values of the search key with pointers to data-file records that have that value for the attribute(s) of the search key.
- It is common to create a primary index on the primary key of a relation and to create secondary indexes on some of the other attributes.



Database Management Systems

Indexes

- A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file.
- The first field is of the same data type as the ordering key field—called the primary key—of the data file, and the second field is a pointer to a disk block (a block address).
- There is one index entry (or index record) in the index file for each block in the data file.
- Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values



Database Management Systems

Example of cost based optimization

Number of records = 30,000
Record length = 100 bytes
Block size = 1024 bytes
Blocking factor for data record= 1024/100 = 10
Number of blocks required to store 30,000 records = 30000/10 = 3000

A binary search of 3000 blocks will require $\log_2 3000 = 12$ block access

Length of the ordering key = 9 bytes
Length of the block address field = 6 bytes
Record length of the index file = 9+6 = 15 bytes

Blocking factor for the index file = 1024/15 = 68
Number of entries in the index file = Number of blocks = 3000
Number of blocks required for storing index information = 3000/68 = 45

A binary search on the index file will require $\log_2 45 = 6$ block access

To retrieve data from the data file will require 1 additional block access for a total of 6 + 1 = 7 block accesses

Database Management Systems

DDL for creating Indexes

```
Create table Customer2 (CustId int not null,
                        CustName varchar(50) not null,
                        Address  varchar(50) not null,
                       City  varchar(50) not null,
                       State  varchar(50) not null,
                       PostalCode  varchar(50) not null,
                       primary key (custid));
```

show index from customer2;

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation
customer2	0	PRIMARY	1	CustId	A

create index CustomerName_Idx on customer2(CustName);

show index from customer2;

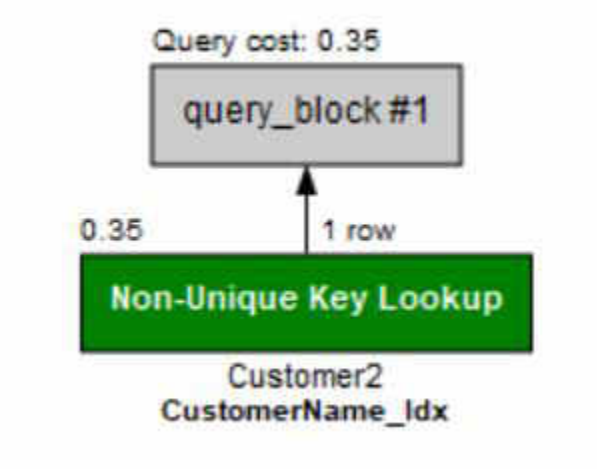
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality
customer2	0	PRIMARY	1	CustId	A	313302
customer2	1	CustomerName_Idx	1	CustName	A	382200

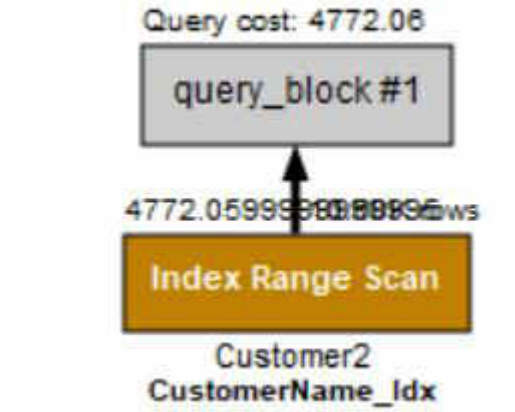
Database Management Systems

Query execution plan

select * from Customer2 where CustName = '829869ABC' ;



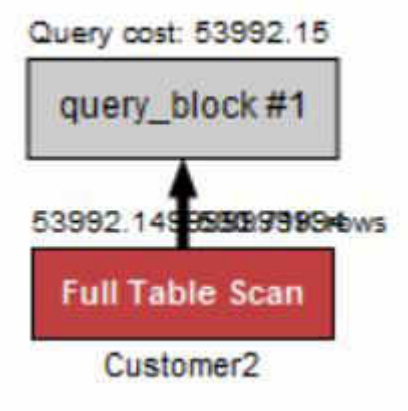
select * from Customer2 where CustName like '95%ABC' ;



Database Management Systems

Query execution plan

select * from Customer2 where City = 'Bengaluru' ;



- Execution plans for SQL statements have a significant impact on the overall performance of a database system.
- The optimizer relies on object statistics (e.g., number of rows, distribution of column values, etc.) and system statistics (e.g., I/O bandwidth of the storage subsystem).
- The optimality of the final execution plan depends primarily on the accuracy of the statistics fed into the cost model.
- To update the object statistics, users can execute the **analyze** command
- analyze table <table name> update histogram on <column name>;
- Example: analyze table customer2 update histogram on CustName;

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

Database Management Systems

NoSQL Databases

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

Database Management Systems

Why NoSQL ??

- Most features of RDBMS are not required in internet/web-based applications.
  - Features like structured data storage, powerful query language, data consistency.
- Instead, other features are more important for internet/web-based application.
  - Semi-structured data, high performance, availability, scalability, replication.

Database Management Systems

What do these applications need ??

- Google/Bing/Yahoo Index of documents on the web
- Google/Yahoo/Hotmail emails
- Amazon Products, Reviews, Ratings, Customers, Addresses
- Facebook Posts(Videos, Images, etc), Likes and Shares
- Twitter Tweets, Replies, Retweets, Likes.
- Flickr, Google Photos
- Youtube/Netflix Videos
- News Sites Articles, comments and replies

Scenario - 01:

- You uploaded a picture to facebook.
  - Some of your friends see the picture and some do not see it. Is it OK?
- You withdrew money from ATM.
  - Is it OK to see two different balances like your friends are seeing two different views of your “facebook wall”?
- Is it better to let some friends view the post or wait till everyone can see the same post?

Scenario - 02:

You are browsing amazon for buying some item.

- Are there more reads or writes?
- Are there only text or text and images and videos of advertisements, product usage and user reviews and ratings as well?
- Are all the products (say a camera and a mobile phone) having the same attributes?
- Are there thousands or millions of – users, products, reviews, ratings, pictures, videos, etc?
- Is it OK to shutdown the servers for maintenance?
- Some time there are thousands of users and some other time there are millions of users.

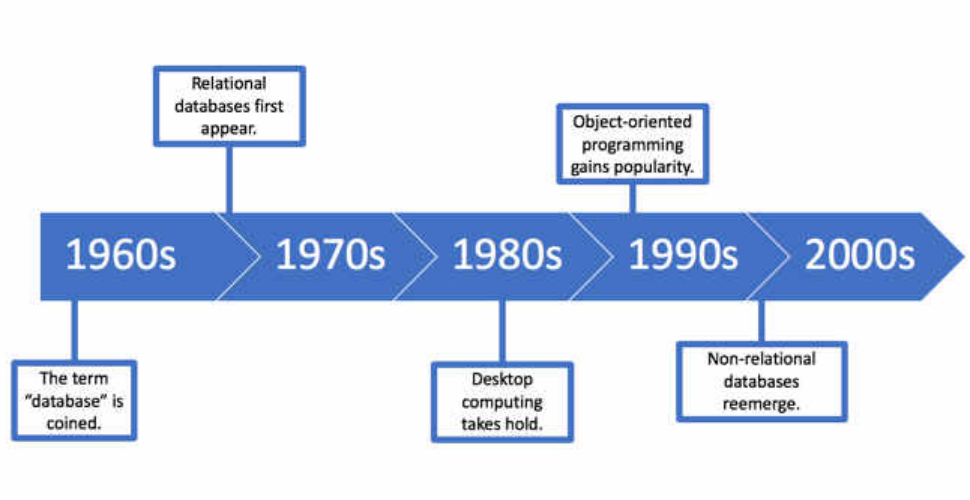
NOSQL Databases

- NoSQL databases emerged when SQL databases could not meet the requirements of new web applications like the ones mentioned earlier.
- NoSQL (non-relational) databases have existed since 1960s.
  - The name NoSQL was given in 1998 (wikipedia).
- Emergence of Internet (Increase in volume of data and the need for a better way of managing the data):
- Search-indexing, Web based Email, Amazon shopping, Social Network: Facebook, Twitter, Flikr, NetFlix, Youtube, ...

Database Management Systems

NoSQL Databases

- NoSQL (not only SQL) is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built.
- NoSQL databases are especially useful for working with large sets of distributed data.
- NoSQL is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats
- **NoSQL databases are databases that store data in a format other than relational tables**



Database Management Systems

NoSQL Databases

- NoSQL stands for not only **SQL**.
  - **SQL** stands for **relational databases** and NOT for **SQL (language)**
- Different applications require different (other than RDBMS) database approaches.
- Some NoSQL databases
  - BigTable by Google (Column-based or Wide Column, Apache Hbase is based on BigTable)
  - DynamoDB by Amazon (Key-Value store, Voldemort is key-value store)
  - Cassandra by Facebook (Based on both key-value and column-based. Available as Apache Cassandra)
  - MongoDB, CouchDB (Document based)
  - Neo4J Graph database.

Database Management Systems

Characteristics of NoSQL systems

- Scalability
  - From 100s to millions of users/usage/objects.
  - Adding more servers and databases handles increased users/usage/objects.
- Availability
  - Always on. Imagine life(for customers and companies) without Google / Amazon /FB...
- Replication Models
  - How and Where to store a copy (in case things go wrong, read from the copy)?
- Sharding of files (Horizontal partitioning of files)
  - Dividing rows into many parts and storing in many servers (for faster access).
- High Performance Data Access
  - The dreaded hourglass/spinning wheel… users will run away from slow sites.
  - hashing or range partitioning on object keys

Database Management Systems

Characteristics of NoSQL systems

- Not requiring a Schema
  - Mostly accessed by keys.
  - Semi-structured data described by JSON or XML
- Less powerful Query Language
  - No support for SQL (or limited SQL support)
  - Access for CRUD through API
- Versioning
  - Storing multiple version of data with time stamps.

Types of NoSQL Databases

- **Document databases** store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
- **Key-value databases** are a simpler type of database where each item contains keys and values.
- **Wide-column** stores store data in tables, rows, and dynamic columns.
- **Graph databases** store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

Database Management Systems

Datamodel and the associated Databases

Key-Value:

- BerkleyDB
- LevelDB
- Memcached
- Redis
- Riak

Document:

- CouchDB
- MongoDB
- Raven
- TerraStore

Columnar:

- Cassandra
- Amazon SimpleDB

Database Management Systems

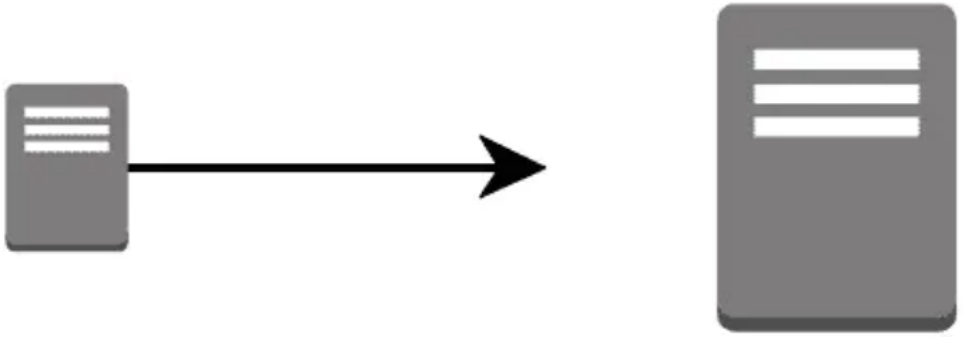
Differences between RDBMS and NoSQL Databases

SQL	NOSQL
Relational Database management system	Distributed Database management system
Vertically Scalable	Horizontally Scalable
Fixed or predifined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries

Database Management Systems

Differences between RDBMS and NoSQL Databases

Vertical Scaling



Vertical scaling refers to increasing the processing power of a single server or cluster. Both relational and non-relational databases can scale up, but eventually, there will be a limit in terms of maximum processing power and throughput. Additionally, there are increased costs with scaling up to high-performing hardware, as costs do not scale linearly.

Differences between RDBMS and NoSQL Databases

Horizontal Scaling

Horizontal scaling, also known as scale-out, refers to bringing on additional nodes to share the load. This is difficult with relational databases due to the difficulty in spreading out related data across nodes. With non-relational databases, this is made simpler since collections are self-contained and not coupled relationally. This allows them to be distributed across nodes more simply, as queries do not have to "join" them together across nodes.

Scaling MongoDB horizontally is achieved through **sharding** (preferred) and **replica sets**.

Database Management Systems

Differences between RDBMS and NoSQL Databases

In a relational database, we'd likely create two tables: one for Users and one for Hobbies.

Users

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

In order to retrieve all of the information about a user and their hobbies, a single document can be retrieved from the database. No joins are required, resulting in faster queries in MongoDB.

{ "_id": 1,
"first_name": "Leslie",
"last_name": "Yepp",
"cell": "8125552344",
"city": "Pawnee",
"hobbies": ["scrapbooking", "eating waffles", "working"] }

Database Management Systems

Categories of NoSQL Databases

Document databases

Store data elements in document-like structures that encode information in formats such as JSON.

+

Common uses include content management and monitoring Web and mobile applications.

+

EXAMPLES:
Couchbase Server, CouchDB, MarkLogic, MongoDB

Graph databases

Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.

+

Common uses include recommendation engines and geospatial applications.

+

EXAMPLES:
Allegrograph, IBM Graph, Neo4j

Key-value databases

Use a simple data model that pairs a unique key and its associated value in storing data elements.

+

Common uses include storing clickstream data and application logs.

+

EXAMPLES:
Aerospike, DynamoDB, Redis, Riak

Wide column stores

Also called table-style databases—store data across tables that can have very large numbers of columns.

+

Common uses include Internet search and other large-scale Web applications.

+

EXAMPLES:
Accumulo, Cassandra, HBase, Hypertable, SimpleDB

Database Management Systems

Comparison of Features of NoSQL Databases

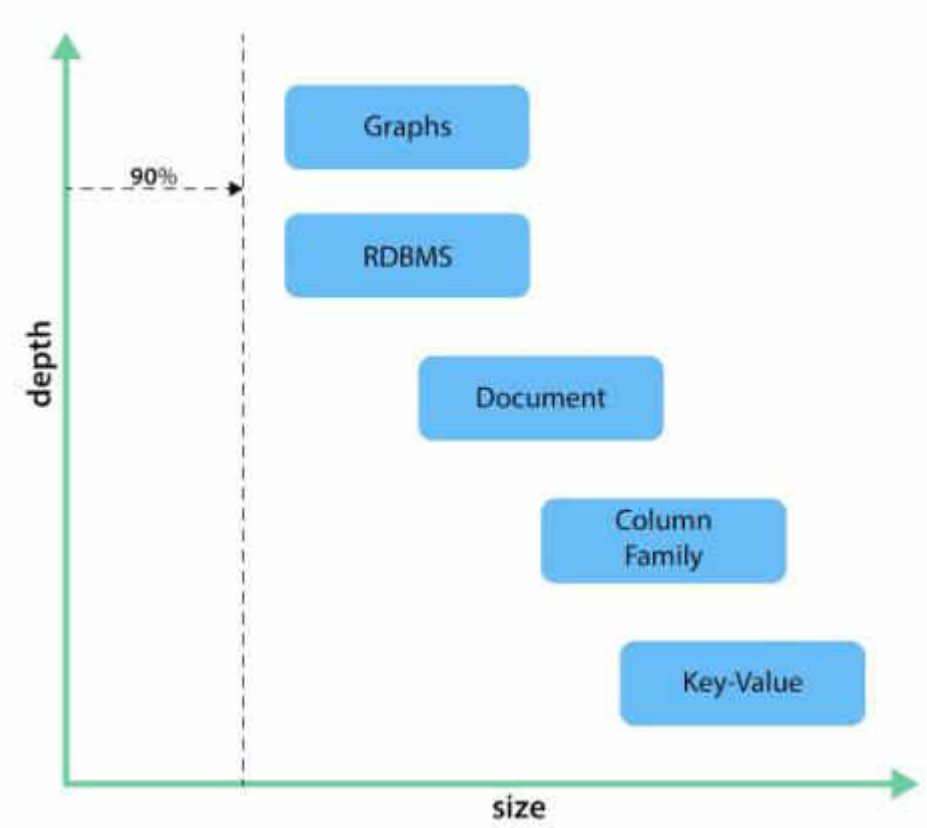
Data model	Performance	Scalability	Flexibility	Complexity	Functionality
Key–value store	high	high	high	none	variable (none)
Column-oriented store	high	high	moderate	low	minimal
Document-oriented store	high	variable (high)	high	low	variable (low)
Graph database	variable	variable	high	high	graph theory
Relational database	variable	variable	low	moderate	relational algebra

Database Management Systems

When to use which database

- Calculating average income
  - Use a **Relational** database.
- Building a shopping cart
  - Use a **key-value** Store.
- Storing structured product information
  - Use a **document** data store
- Describing how a user got from point A to point B
  - Use a **graph** database.

Database type stacks wrt measuring depth and size



THANK YOU

Department of Computer Science and Engineering

DATABASE TECHNOLOGIES

NoSQL Databases - DocumentDB

Mahitha G

Department of Computer Science and Engineering

DATABASE TECHNOLOGIES

Document Database - MongoDB

Mahitha G
Department of Computer Science and Engineering

DATABASE TECHNOLOGIES

MongoDB - Hu(mongo)us

- MongoDB was designed as a scalable database - the name Mongo comes from “humongous”- with performance and easy data access as core design goals.
- It is a document database, which allows you to store objects nested to whichever depth you’d like and you can query that nested data in an ad-hoc fashion.
- It enforces no schema, so documents can contain fields or types that no other document in the collection contains
- MongoDB hits a sweet spot between the powerful query ability of a relational database and the distributed nature of other databases
- Mongo is a JSON document database. A Mongo document is similar to a relational table row without a schema, whose values can nest to an arbitrary depth

DATABASE TECHNOLOGIES

MongoDB

- Document-Oriented storage
- Full Index Support
- Replication & High Availability
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Map/Reduce functionality

```
{
   "_id" : ObjectId("4d0b6da3bb30773266f39fea"),
   "country" : {
      "$ref" : "countries",
      "$id" : ObjectId("4d0e6074deb8995216a8309e")
   },
   "famous_for" : [
      "beer",
      "food"
   ],
   "last_census" : "Sun Jan 07 2018 00:00:00 GMT -0700 (PDT)",
   "mayor" : {
      "name" : "Ted Wheeler",
      "party" : "D"
   },
   "name" : "Portland",
   "population" : 582000,
   "state" : "OR"
}
```

JSON document

MongoDB

MongoDB stores data as documents.

A document in MongoDB consists of field-value pairs.

Documents are organized in a structure called collection.

To make an analogy, document can be considered as a row in a table and collection can be considered as an entire table.

DATABASE TECHNOLOGIES

MongoDB – Who uses????

DATABASE TECHNOLOGIES

MongoDB use case

- **SEGA** uses MongoDB for handling 11 million in-game accounts.
- **Cisco** moved its VSRM (video session and research manager) platform to Couchbase to achieve greater scalability.
- **Aer Lingus** uses MongoDB with Studio 3T to handle ticketing and internal apps.
- Built on MongoDB, The Weather Channel's iOS and Android apps deliver weather alerts to 40 million users in real-time.

MongoDB – Data model

MongoDB provides two types of data models: —

Embedded data model

Normalized data model.

Based on the requirement, use either of the models

MongoDB – Data model

Embedded Data Model:

This model, can embed or have all the related data in a single document, it is also known as de-normalized data model.

For example, the details of employees in three different documents namely, Personal_details, Contact and, Address, can be embed in a single one

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
```

DATABASE TECHNOLOGIES

MongoDB – Data mo

Normalized Data Model:

This model, can refer the su

For example, re-write the pr

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

MongoDB - Commands

Use Command

The command will create a new database if it doesn't exist, else returns the existing database (connects to the new database created)

Format: use database_name

Example: use testdb

//testdb is the current db on which you can work

MongoDB - Commands

db Command: To check which database you are connected

Example: db

Testdb

//output – name of the db displayed.

Command to list all databases

show dbs

//testdb not shown here:

// database should have atleast one document in it to be displayed

MongoDB - Commands

dropdatabase Command: To drop the current database

Format: db.dropDatabse()

Example: db.dropDatabase ()

// the current database is dropped ie Testdb is dropped

MongoDB - Datatypes

Some of the Datatypes supported

String
Integer
Boolean
Double
Arrays
Timestamp
Object
Symbol
Date
Code
Binary Data
Regular Expression
Null

DATABASE TECHNOLOGIES

MongoDB - Commands

Collections: Each database has a collection. MongoDB stores documents in collections. Collections are analogous to tables in relational databases

db.createCollection is used to create collection.

Format: **db.createCollection(name, options)**

name indicates name of the collection to be created.

options is a document and is used to specify configuration of collection.

Options parameter is optional

Example: **db.createCollection("myTest")** **// myTest is the collection created under testdb**

To list all collections in the database:

```
show collections
```

//will list out all the collections in the database

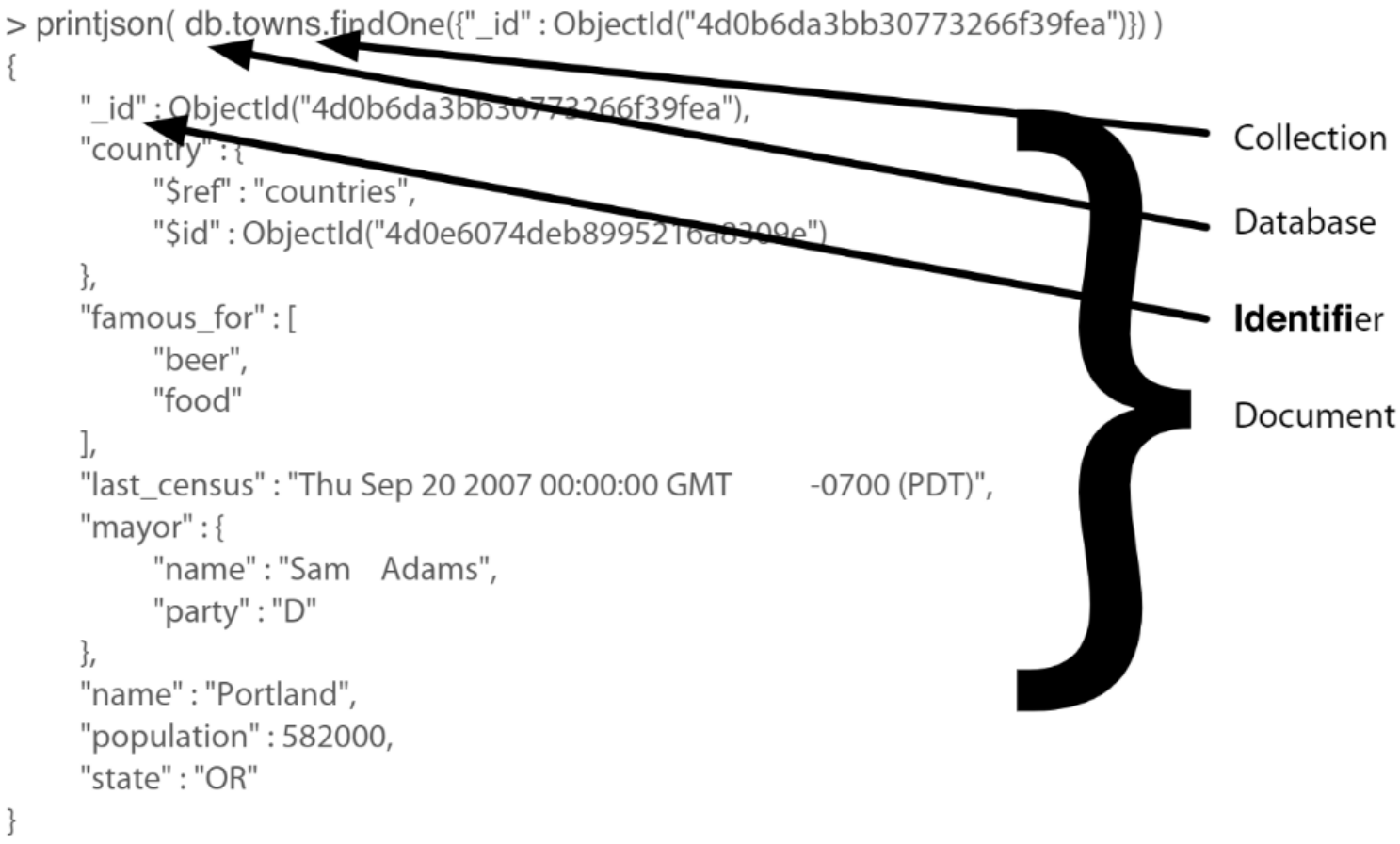
To create a collection and add a document into the collection:

```
db.sample.insert({“Name”:”Rama”})
```

//creates a new collection by name “sample” and inserts a document into it

DATABASE TECHNOLOGIES

Mongodb document printed as JSON



To drop a collection

Format: db.Collection_name.drop()

Example: db.sample.drop()

Two options to insert a document into a collection

- **Insert one document at a time**
- **Insert multiple documents through an array**

Format: db.collection_name.insert(document) // insert one document @ a time

db.collection_name.insert([comma separated documents]) //insert multiple

Example: db.sample.insert("Name" : "Rama")

db.sample.insert("Name":"Rama", "Name":"John","Name":"Praveen")

// other variations: db.collection.insertone(), db.collection.insertMany()……

// if sample collection does not exist, it creates and then add's the document.
// if _id parameter is not specified – mongo adds _id which is a unique identifier

Find() – method to query documents from a collection

Format: db.collection.find(query, projection)

//displays the documents in unstructured format

Example: db.sample.find()
// displays all documents with all fields

db.sample.find().pretty()

// displays the documents in structured format

Parameter	Type	Description
Query	Document	• Optional. Specifies selection filter. • To return all documents omit this parameter or pass an empty document ({})
Projection	Document	• Optional. Specify the fields to be returned. • To return all fields in the matching documents, omit this

Query: Display the first document in the collection

db.employee.findone()

//findone() does not support pretty() method

Query: Display the document of employee with empid=2

db.employee.find({“empid”:2})

Query: Display the document related to young employees (age<30)

db.employee.find({“age”:{$lt:”30”}})

lt -> less than

gt -> greater than

lte -> less than or equal to

gte -> greater than or equal to

Query: Return the employees born after '1988-10-10' from employee collection

```
db.employee.find( { "birth": { $gt: new Date('1950-01-01') } } )
```

Query : Return all the employees who have skilled in mongodb

```
db.employee.find({"skill":"mongodb"})
```

Query: Return all the employees skilled in mongodb and salary of 80000

```
db.employee.find({"skill":"mongodb","salary":"80000"})
```

Query: Return all employees either skilled in mongodb or salary of 80000

```
db.employee.find({$or:[{"skill":"mongodb"},{"salary":"80000"}])
```

```
//and -> , or $and [{condition 1},{condition 2}]
//or -> $or:[{condition 1},{condition 2}]
```

Query: Return documents where birth is between 1940-01-01 and 1960-01-01

```
db.employees.find( { birth: { $gt: new Date('1940-01-01'), $lt: new Date('1960-01-01') } } )
```

Query: Return documents where employees are skilled in “mongodb” and either with salary 80000 or 85000

```
db.employee.find({“skill”:”mongodb”}, $or:[{”salary”:”80000”}, {”salary”:”85000”}])
```

Update values in documents:

Format: db.collection_name.update(selection_criteria, update_value)

Query: update salaries of all employees with skill “mongodb”

db.employee.update({“skill”:”mongodb”},{$set:{“salary”:”1000000”}})

// modifies one document

// to update all

db.employee.update({“skill”:”mongodb”},{$set:{“salary”:”1000000”}}) {multi:true}

To remove documents from the collection

Format: db.collection_name.remove(selection_criteria,flag)

db.collection_name.remove({}) // removes all documents from the collection

Query: Remove all employees skilled in “mongodb”

db.employee.remove({“skill”:”mongodb”},1}) // removes only one employee

db.employee.remove({“skill”:”mongodb”}}) // removes all employees skilled in mongodb

To display only necessary data from documents in the collection

Format: db.collection_name.find(selection_criteria, fields required)

db.employee.find({},{“firstname”:1})

//display firstname and object_id of all the employees

{} -> all documents ie no selection criteria
1 -> indicates display firstname (select firstname)
0 -> indicates do not select the field

db.employee.find({},{“firstname”:1,”_id”:0})

//displays only firstname of all the employees

Add these to the previous query:

.limit(5) -> it displays only first 5 instances

.skip(3) -> skips first 3 instances

.skip(3).limit(5) ->skip first 3 and from there display 5

.sort({“firstname”:1}) -> sorts according to firstname

1 -> ascending order
-1 -> descending order

MongoDB - Aggregation

Foramt: db.collection_name.aggregate(aggregate_operation)

Query: To check how many male and female employees:

db.employee.aggregate([{$group:_id:"$gender", Total : "$sum":1}])

$sum, $avg, $max, $min

db.collection_name.count() // to know number of documents in the collection

DATABASE TECHNOLOGIES

Difference between Mongodb and RDBMS

RDBMS	MongoDB
It is a relational database.	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage.
It is vertically scalable i.e increasing RAM.	It is horizontally scalable i.e we can add more servers.
It has a predefined schema.	It has a dynamic schema.
It is quite vulnerable to SQL injection.	It is not affected by SQL injection.
It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).	It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.
It is column-based.	It is field-based.
It does not provide JavaScript client for querying.	It provides a JavaScript client for querying.
It supports SQL query language only.	It supports JSON query language along with SQL.

THANK YOU

Mahitha G
Department of Computer Science and Engineering

mahithag@pes.edu

Database Management Systems

NoSQL – Key Value Store

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science
and Engineering

Database Management Systems

Key-Value database

- The key-value store is the simplest NoSQL model
- There is no query language – but set of operations
- The key is a unique identifier associated with a data item and is used to locate this data item rapidly.
- The value is the data item itself, and it can have different formats for different key-value storage systems.

Example:

- A file system could be considered a key-value store if you think of the file path as the key and the file contents as the value

Examples of Key-Value Stores

Phone Directory	• **Key :**BOB • **Value**: (123) 456-7890
Artist Info	• **Key :**artist:1:name • **Value :**JM
IP Forwarding Table	• **Key :** 202.45.12.34 • **Value :** 01:23:36:0f:a2:33
Stock Trading	• **Key :** 234567890 • **Value :**CERN, Sell, 50, 52.78

Database Management Systems

Examples of Key-Value Stores

Database Management Systems

Key Value Store - Usecases

- Storing Session Information
- User Profiles, preferences
- Shopping Cart Data
- Article/Blog Comments
- Product Categories/Details/Reviews
- Internet Protocol Forwarding tables
- Telecom directories

Database Management Systems

When not to Use Key Value Store

Relationships among Data

Multi-operation Transactions

Query by Data

What Type of Data can be stored in a Key-Value Database?

The Key

The key in a key-value pair must (or at least, should) be unique. This is the unique identifier that allows you to access the value associated with that key.

The Value

The value in a key-value store can be anything, such as text (long or short), a number, code such as HTML, programming code such as PHP, an image, etc.

The value could also be a list, or even another key-value pair encapsulated in an object.

DynamoDB

- DynamoDB is a cloud-based database available through Amazon Web Services (AWS)
- The basic data model in DynamoDB uses the concepts of tables, items, and attributes.
- A table in DynamoDB does not have a schema; it holds a collection of self-describing items.
- Each item will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued.
- When a table is created, it is required to specify a table name and a primary key.

Primary key can be of two types:

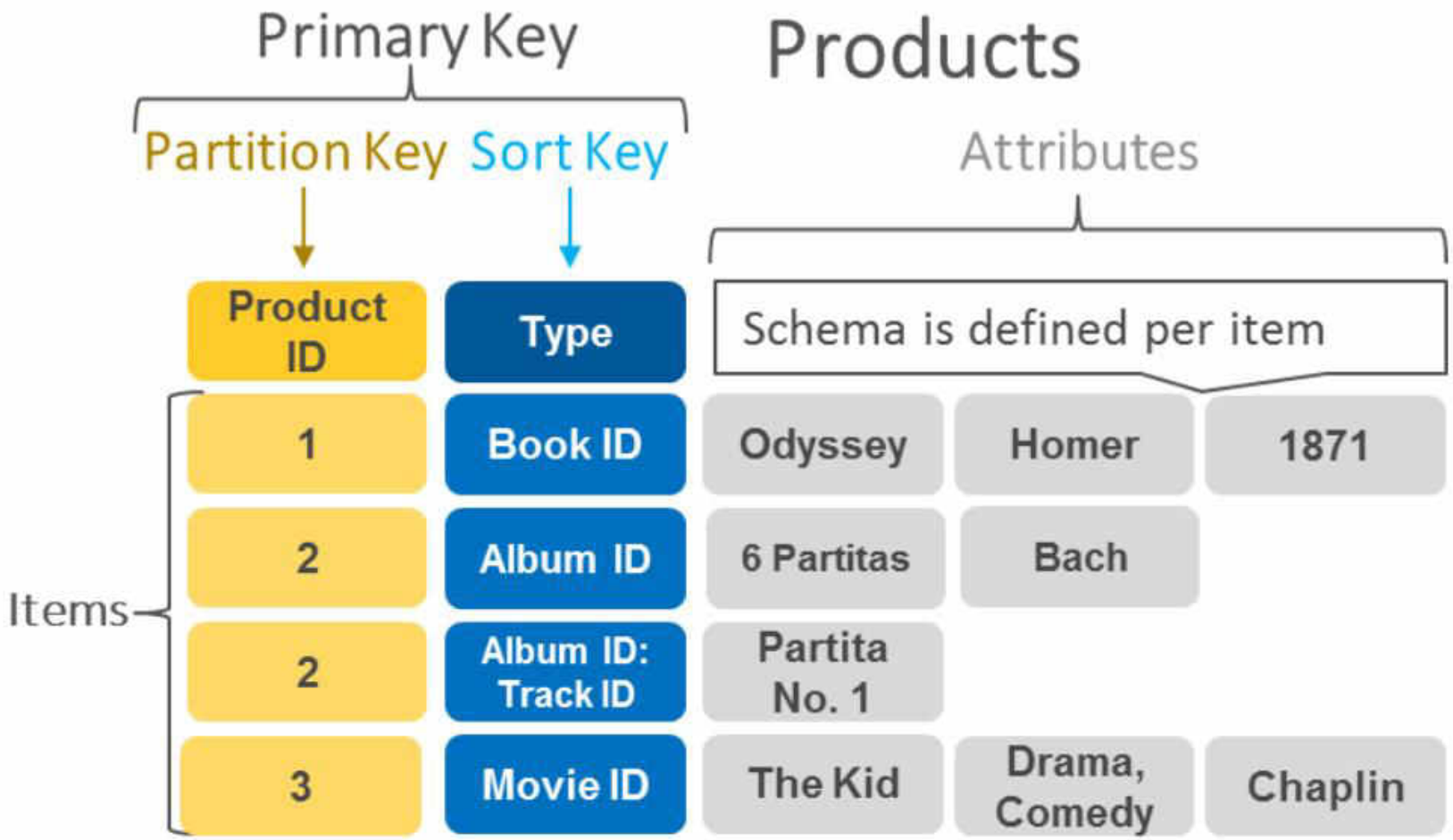
- **Single Attribute:** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a hash type primary key.
- **Pair of Attributes:** This is called a hash and range type primary key. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value.
- https://www.youtube.com/watch?v=DQnjNgixZQs
- https://www.youtube.com/watch?v=a_LaXOzG0co

DynamoDB Features

- SSD's for storage
- Built in High Availability
- Predictable Performance
- Massive Scalability
- Serverless Apps
- Read Heavy Apps

Database Management Systems

An example of data stored as key-value pairs in DynamoDB.



THANK YOU

Department of Computer Science and Engineering

Database Management Systems

NoSQL – Column Oriented DB

Prof. Bhargavi M & Prof. Vidhyashree DM

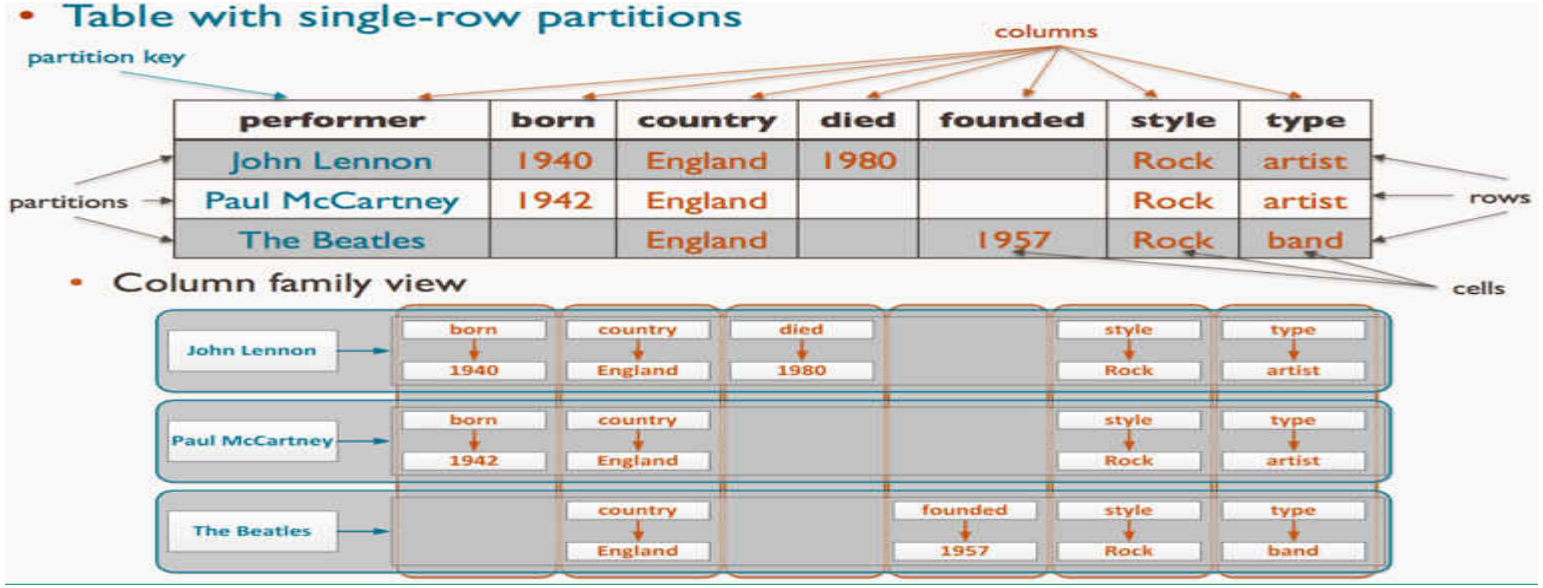
Department of Computer Science
and Engineering

Database Management Systems

Wide-Column Store

A wide-column stores use columns to store data. It can group related columns into column families. Individual rows then constitute a column family.

A typical column family contains a row key as the first column, which uniquely identifies that row within the column family. The following columns then contain a column key, which uniquely identifies that column within the row, so that you can query their corresponding column values.



Wide-Column Store

Examples of Column Databases

- Cassandra – Free, open-source
- HBase – Free, open-source

Use Cases

- Column stores offer very high performance and a highly scalable architecture. Because they're fast to load and query, they've been popular among companies and organizations dealing with big data, IoT, and user recommendation and personalization engines.
- **Spotify** uses Cassandra to store user profile attributes and metadata about artists, songs, etc. for better personalization.
- **Facebook** initially built its revamped Messages on top of HBase, but is now also used for other Facebook services like the Nearby Friends feature and search indexing.
- **Outbrain** uses Cassandra to serve over 190 billion personalized content recommendations.

Benefits of Column Store Databases

- **Benefits of Column Store Databases**

Some key benefits of columnar databases include:

- **Compression**. Column stores are very efficient at data compression and/or partitioning.
- **Aggregation queries**. Due to their structure, columnar databases perform particularly well with aggregation queries (such as SUM, COUNT, AVG, etc).
- **Scalability**. Columnar databases are very scalable. They are well suited to massively parallel processing, which involves having data spread across a large cluster of machines – often thousands of machines.
- **Fast to load and query**. Columnar stores can be loaded extremely fast. A billion row table could be loaded within a few seconds. You can start querying and analysing almost immediately.

Database Management Systems

Hbase – column based or Wide Column NoSQL system

- HBase is a column-oriented database that prides itself on its ability to provide both consistency and scalability.
- HBase is a Apache product and is inspired by BigTable 's white paper and it is open source.
- HBase stores data in buckets it calls tables, which contain cells that appear at the intersection of rows and columns.
- Hbase tables don’t behave like relations, rows don’t act like records, and columns are completely variable and not enforced by any predefined schema
- HBase has some built-in features that other databases lack, such as versioning, compression, garbage collection (for expired data) and in-memory tables
- HBase guarantees atomicity at the row level, which means that you can have strong consistency guarantees at a crucial level of HBase’s data model

Database Management Systems

HBase

- HBase is designed to be fault tolerant. Hardware failures may be uncommon in individual machines but, in large clusters, node failure is the norm (as are network issues).
- HBase can gracefully recover from individual server failures because it uses both write-ahead logging, which writes data to an in-memory log before it's written (so that nodes can use the log for recovery rather than disk), and distributed configuration, which means that nodes can rely on each other for configuration rather than on a centralized source.
- HBase is part of the Hadoop ecosystem, where it benefits from its proximity to other related tools
- HBase supports three running modes:
    - Standalone mode is a single machine acting alone.
    - Pseudo-distributed mode is a single node pretending to be a cluster.
    - Fully distributed mode is a cluster of nodes working together.

Hbase data model:

The data model in Hbase organizes data using the concepts of namespaces, tables, column families, column qualifiers, columns, rows, and data cells.

A column is identified by a combination of (column family:column qualifier).

Data is stored in a self-describing form by associating columns with data values, where data values are strings.

Hbase also stores multiple versions of a data item, with a timestamp associated with each version unique keys are associated with stored data items for fast access

Hbase data model:

- Tables and Rows
- Column Families, Column Qualifiers, and Columns
- Versions and Timestamps
- Cells
- Namespaces

Database Management Systems

Examples in Hbase

- **Creating a table EMPLOYEE with column families: Name, Address, and Details**

create ‘EMPLOYEE’, ‘Name’, ‘Address’, ‘Details’

- **Inserting some in the EMPLOYEE table**

put ‘EMPLOYEE’, ‘row1’, ‘Name:Fname’, ‘John’
put ‘EMPLOYEE’, ‘row1’, ‘Name:Lname’, ‘Smith’
put ‘EMPLOYEE’, ‘row1’, ‘Name:Nickname’, ‘Johnny’
put ‘EMPLOYEE’, ‘row1’, ‘Details:Job’, ‘Engineer’
put ‘EMPLOYEE’, ‘row1’, ‘Details:Review’, ‘Good’
put ‘EMPLOYEE’, ‘row2’, ‘Name:Fname’, ‘Alicia’
put ‘EMPLOYEE’, ‘row2’, ‘Name:Lname’, ‘Zelaya’
put ‘EMPLOYEE’, ‘row2’, ‘Name:MName’, ‘Jennifer’
put ‘EMPLOYEE’, ‘row2’, ‘Details:Job’, ‘DBA’
put ‘EMPLOYEE’, ‘row2’, ‘Details:Supervisor’, ‘James Borg’

- Some CRUD operations of Hbase

Database Management Systems

Examples in Hbase

Some CRUD operations of Hbase

Creating a table: create <tablename>, <column family>, <column family>, …
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>

THANK YOU

Department of Computer Science and Engineering

Database Management Systems

NoSQL - Graph Database – Neo4J

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science
and Engineering

Database Management Systems

Graph database

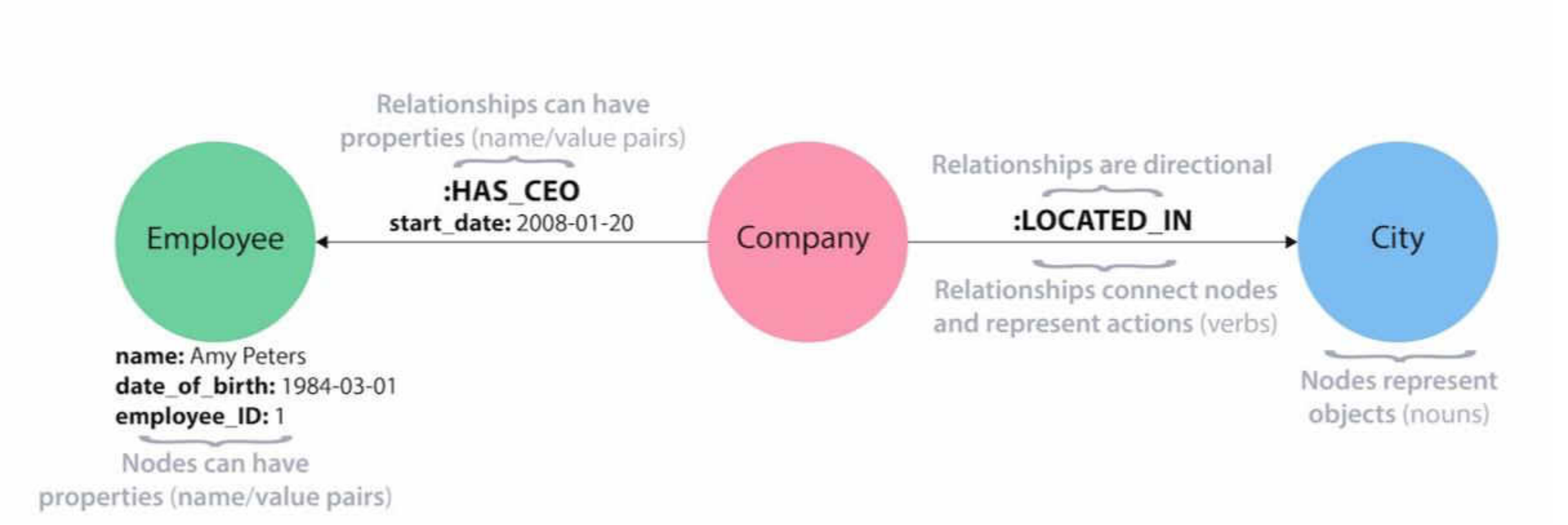
- A graph database is a database designed to treat the relationships between data as equally important to the data itself.
- It is intended to hold data without constricting it to a pre-defined model. Instead, the data is stored like we first draw it out - showing how each individual entity connects with or is related to others
- While RDBMS compute relationships at query time through expensive JOIN operations, a graph database stores connections alongside the data in the model.
- Accessing nodes and relationships in a native graph database is an efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core.

Graph Model

- ***Nodes*** are the entities in the graph. They can hold any number of attributes (key-value pairs) called *properties*. Nodes can be tagged with *labels*, representing their different roles in your domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes.
- ***Relationships*** provide directed, named, semantically-relevant connections between two node entities (e.g. Employee *WORKS_FOR* Company). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction.

Database Management Systems

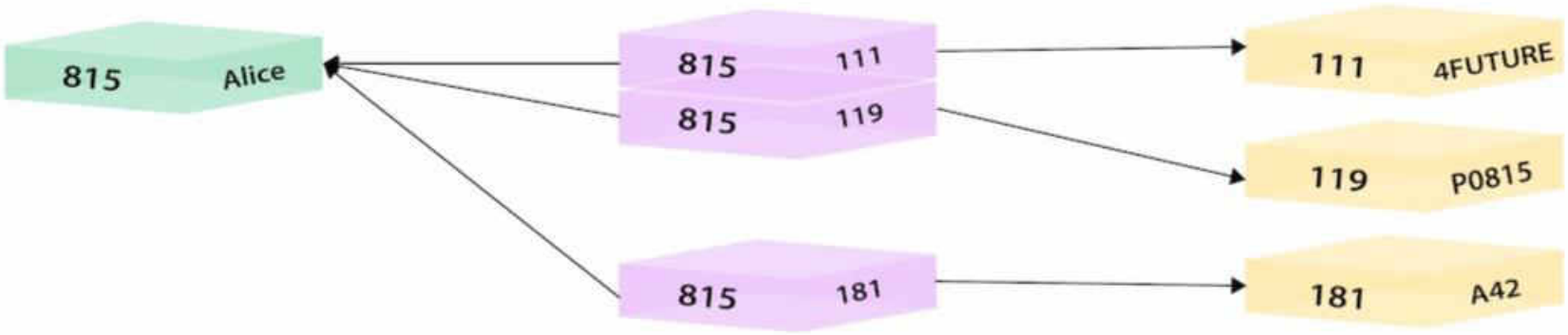
Graph Model Example



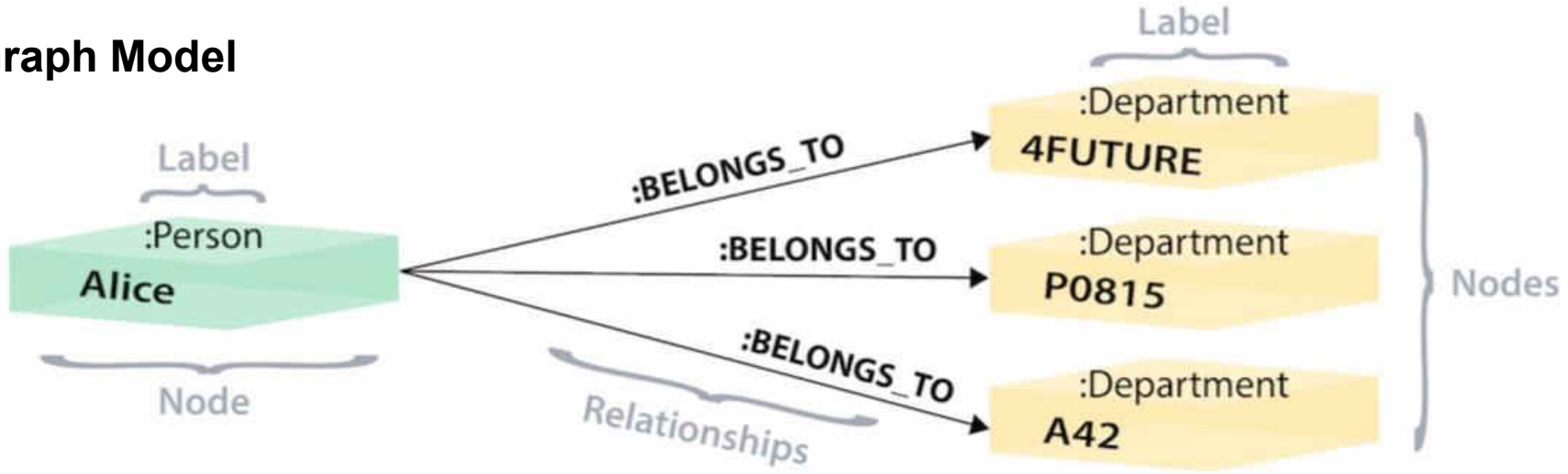
Database Management Systems

Relational vs Graph Model Example

Relational Model



Graph Model



Graph Databases

ArangoDB,

Oracle NoSQL database

Neo4j

OrientDB

AllegroGraph

BlazeGraph

Dgraph and many more….

Database Management Systems

Neo4J

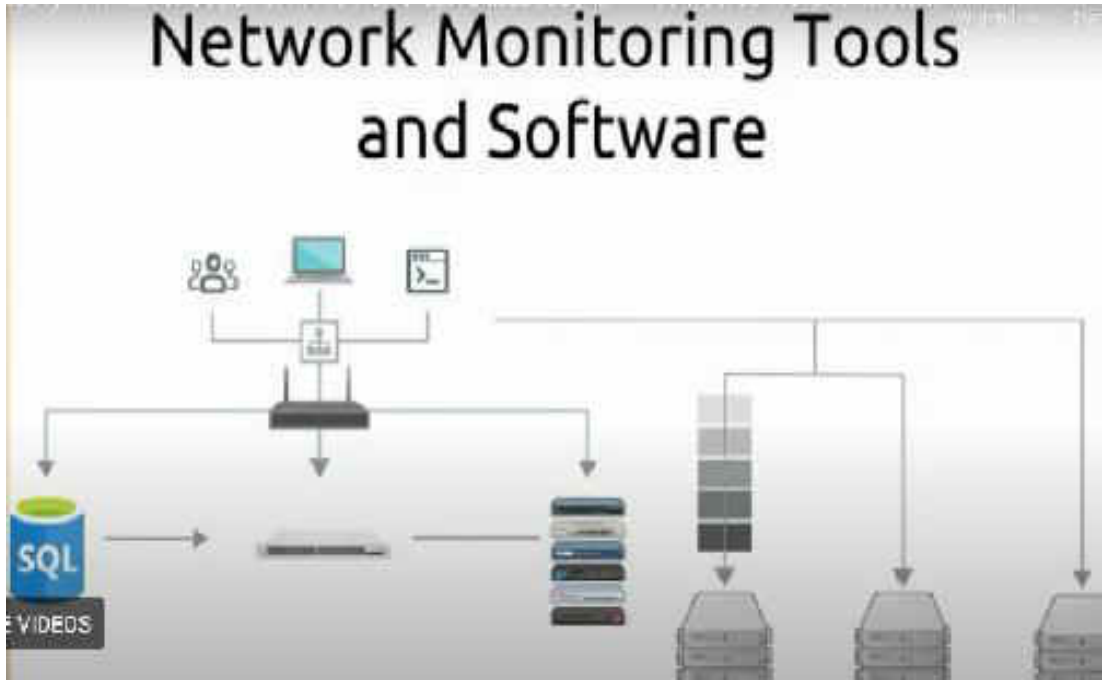
Neo4J is the most popular graph database used by: Adobe, Cisco, Glassdorr, Huawei, HP….

Neo4j is the world's leading open source Graph Database which is developed using Java technology. It is highly scalable and schema free (NoSQL).

Neo4J uses Graph model: Nodes and Relationships (to connect nodes)

Fraud detection and analytics – to detect financial crimes through credit cards, ecommerce and money laundering,

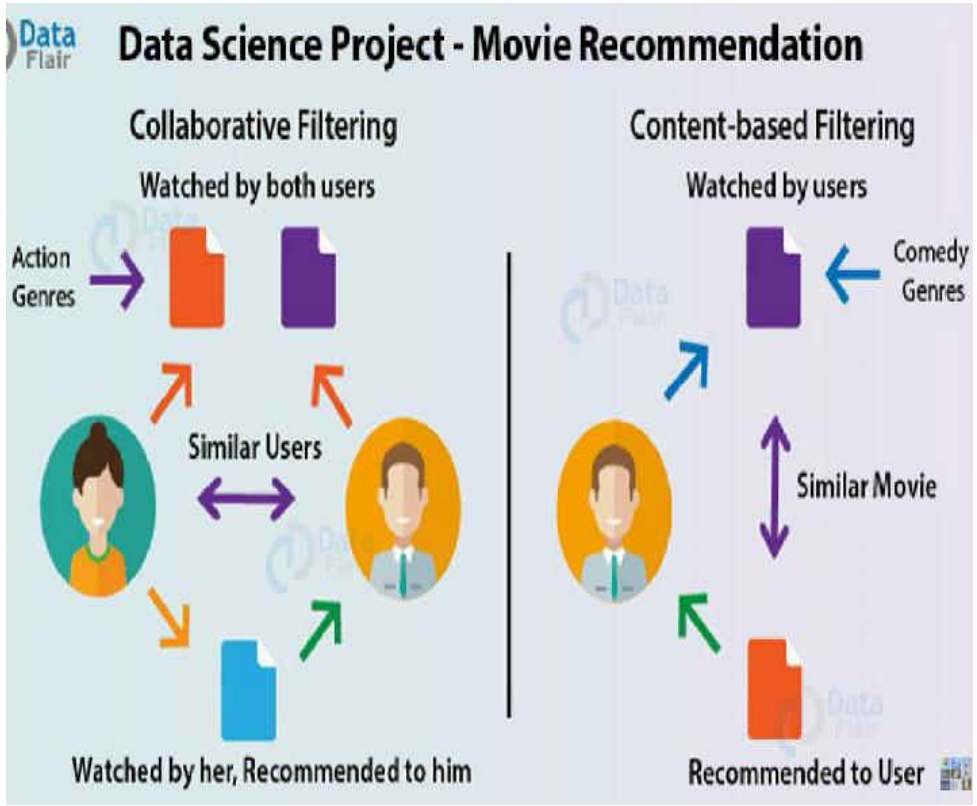
Network Monitoring Tools and Software

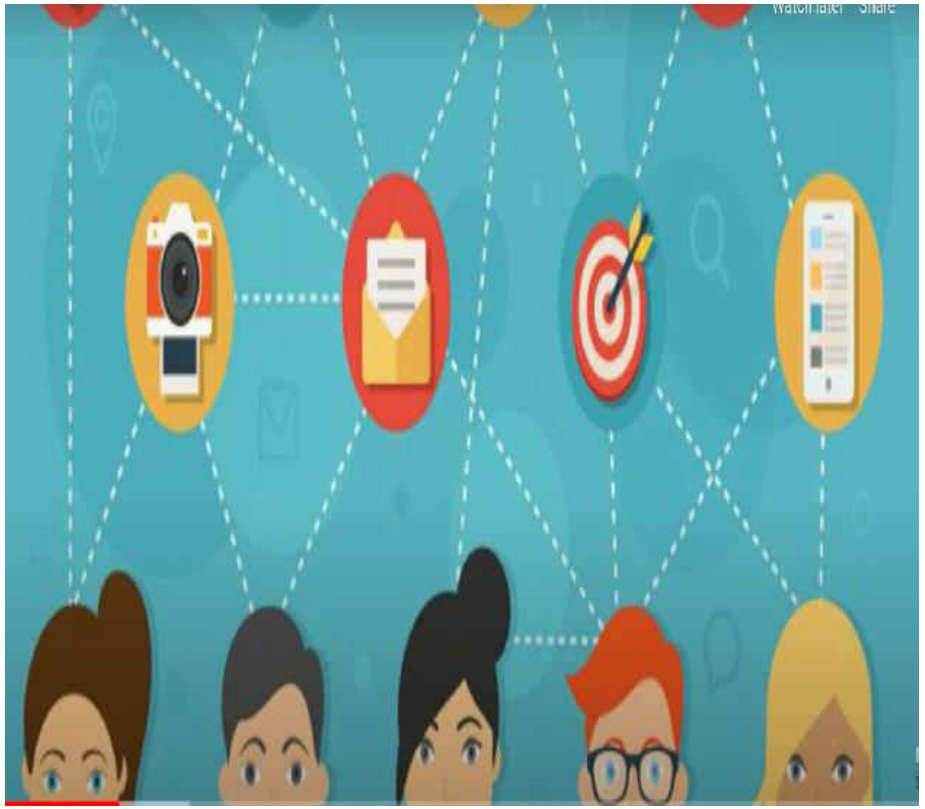


Database Management Systems

Neo4j – Graph DB - Use cases

Recommendation Engines and Social Networking





Database Management Systems

Organisations using Neo4j

Walmart

Ebay

Adbode

Nasa

Microsoft

IBM

Airbnb

Database Management Systems

RDBMS / Graph DB

RDBMS is strictly structured, with tables, rows and columns

Developers and Applications should adhere to this strict table structure

Need complex queries with are memory and computational intensive

Understanding the structure in huge database is challenging

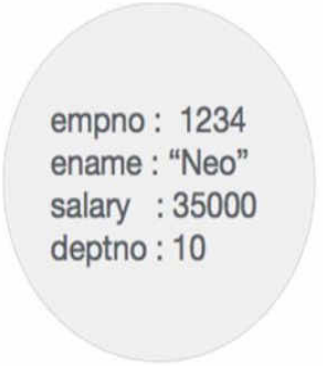
So….use graph database
Graph has nodes and realtionship, each node has properties
Social network is a good example of graph, people are nodes with properties name, age, dob and can play roles like manager, employee, friend…… These roles represents relationship
Graph db uses query language called CQL (cypher query language) which was originally developed for Neo4J
Suitable when data is highly interconnected

Database Management Systems

Neo4j building blocks

Node is a fundamental unit of a Graph. It contains properties with key-value pairs as shown in the following image.

Node Name = "Employee" and it contains a set of properties as key-value pairs.



Employee Node

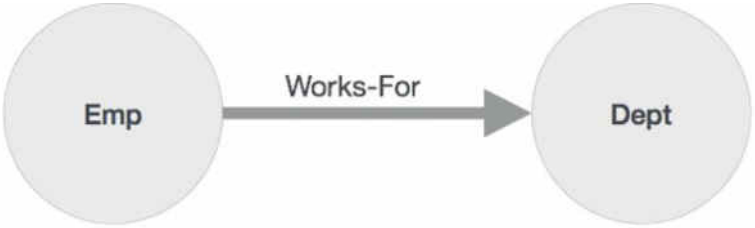
Property is a key-value pair to describe Graph Nodes and Relationships.

Key=Value

Where Key is a String and Value may be represented using any Neo4j Data types.

Relationships connects nodes, Emp and Dept are two different nodes.
"WORKS_FOR" is a relationship between Emp and Dept nodes. The arrow mark from Emp to Dept, describes relationship. Emp worksfor Dept

Label associates a common name to a set of nodes or relationships. relationship can contain one or more labels.
Emp, Dept and worksfor are labels



Oracle DB (RDBMS) has query language SQL, Neo4j has CQL as query language.

CQL stands for Cypher Query Language.

Neo4j CQL:

Is a query language for Neo4j Graph Database.

Is a declarative pattern-matching language.

Follows SQL like syntax.

Syntax is very simple and in human readable format

Database Management Systems

Neo4J – Create and View a Single Node

Create a node:

```
Create (n)
```

// creates a single node
// n is an identifier and not the name of the node.

Output: Created one node in “xyz” ms

View a node:

```
Match(n) return(n);
```

//Fetches a node created in the database
//node is printed with internal property: “id”
// id is used for internal usage, starts from zero and each node will have only one internal id and unique in the database and auto generated by database

Neo4J – Create and View Multiple Nodes

Create multiple nodes:

Create (n) , (m)
// creates two nodes successfully. n and m are identifiers/variables
Output: Created two nodes in “xyz” ms

View all nodes created:

Match(n) return n;
//Fetches the nodes created in the database
Output: 3 nodes are displayed including the one created in previous slide

View some nodes:

Match(n) return n limit 2
//Restrict the number of nodes by using limit clause Output: Displays first 2 nodes with id 0 and 1

Neo4J – Search Nodes by id's

Search a node by internal id:

```
match (n) where id(n) =1 return n
```

// id built in function of Neo4j, returns the internal id of the node when used with where clause
Output: Displays the node with internal id=1

Comparison operator to filter out the nodes (<,>,<=,>=,<>):

To fetch the node with internal id <=6

```
Match(n) where id(n)<=6 return n
```

Output: Displays all 7 nodes (node with id =0,1,2,3,4,5,6)

Search multiple nodes with internal id's (using in clause)

```
Match(n) where id(n) in [1,2,0]  return n
```

Output: Displays 3 nodes with id 0, 1, 2

Database Management Systems

Neo4J – Delete nodes and Database

Delete a node by its internal id:

```
match (n) where id(n) =1 delete n
```

// deletes the node that matches the where clause ie deletes a node with id=1
Output: Deleted one node in “xyz” ms

Delete multiple nodes with id:

```
Match(n) where id(n) in [2,3] delete n
```

// two nodes are deleted with internal id 2 and 3
Output: Deleted two nodes in “xyz” ms

Delete all nodes

```
Match(n) delete n
```

//delete all the nodes in the database if there are no relationships in the graph
Output: Deleted “x” nodes in “xyz” ms

Neo4J – Delete nodes and Database (contd..)

Deleting entire DB:

sudo rm –rf /var/lib/neo4j/data/*

// delets entire graph directory

//all the data is stored in the graph director, so need to delete the graph directory.

// Before this stop the neo4j server using command sudo service neo4j stop

// later restart the server and check in neo4j graphical interface that no data is available in db,

can use neo4j from scratch.

Sudo service neo4j start -> to start the server

Sudo service neo4j stop -> to stop the server

Sudo service neo4j status -> to check the status of the server (running/stopped)

Database Management Systems

Neo4J – Create and Search nodes with labels

Labels are the tags that can be associated to nodes, labels can be used to group the nodes

Labels are optional and each node can have multiple labels

Any valid Unicode string can be used as a label

Creating a node with a label:

Create (n:Person)
Output: Added 1 label, created 1 node, statement executed in “xyz” ms

// creates a node with label Person
// label is represented by a colon followed by name of the label and can be applied only to nodes.
// by convention use Camel case

Database Management Systems

Neo4J – Create and Search nodes with labels

Search node with a labels:

```
Match (n) where n:Person return n
```

// returns a node with label Person
// displays both the id and label of the node

Create a node with multiple labels:

```
create (n : Person : Indian)
```

Output: Added 2 labels, created 1 node, statement executed in “xyz” ms
// a node is created with two labels (person and Indian)
// a node can have multiple labels and a label can have multiple nodes
// a node can have multiple labels as it can represent multiple things at the same time

Neo4J – Create and Search nodes with labels (Contd..)

Search node by multiple labels:

```
Match (n) where n:Person:Indian return n
```

// returns all the nodes with label Person and Indian
// displays both the id and label of the node

Search a node with Person or Indian label or both:

```
Match (n) where n :Person or n :Indian return n
```

//returns nodes with the label Person or with the label Indian or both

Neo4J – Create and Search nodes with labels (Contd..)

Search node by multiple labels:

```
Match (n) where n:Person:Indian return n
```

// returns all the nodes with label Person and Indian
// displays both the id and label of the node

Search a node with Person or Indian label or both:

```
Match (n) where n :Person or n :Indian return n
```

//returns nodes with the label Person or with the label Indian or both

Neo4J – Add Labels to node

Add label to an existing node using set clause:

```
Match (n) set n:Employee return n
// add's a label Employee to all the nodes in the database
```

Add multiple labels to all nodes in database

```
Match (n) set n:Vacation:Food return n
// add's a label Vacation and Food to all the nodes in the database
```

Add label to a single selected node by using id:

```
Match (n) where id (n) =1 set n:Manager return n
//add's a label Manager to the node with the internal id=1
```

Neo4J – Add Labels to node (Contd..)

Add label to multiple selected nodes:

```
Match (n) where id (n) in [2,3] set n: TeamLeader return n
```

//add's a label "TeamLeader" to the nodes with internal id 2 and 3.

Database Management Systems

Neo4J – Remove/Update Labels to node (Contd..)

Remove a label from all nodes in database:

```
Match (n) remove n:Person return n
```

//remove's label "Person" from all the nodes

Remove a label from selected nodes:

```
match (n) where id(n) in [2,3] remove n:Employee return n
```

//remove's label "Employee" from the nodes with id 2 and 3.

Remove multiple labels from all nodes in database:

```
Match (n) remove n:Food:Vacation return n
```

//remove's label Food and vacation from all the nodes which has both or either of these labels

Neo4J – Remove/Update Labels to node (Contd..)

Update a label of a node:

```
Match (n) where id(n) = 0 remove n:Manager set n:Director return n
//update's label of the node with internal id 0 from Manager to Director and return the node
```

List all labels of all nodes:

```
match (n) return distinct labels(n);
// display's all labels existing in db
// distinct removes duplicate labels  and node labels are case sensitive
```

Listing all labels for a node:

```
Match (n) where id(n)=0 return distinct labels(n)
//list all the labels associated with node with id=0
```

Neo4J – Remove/Update Labels to node (Contd..)

Update a label of a node:

```
Match (n) where id(n) = 0 remove n:Manager set n:Director return n
```

//update's label of the node with internal id 0 from Manager to Director and return the node

List all labels of all nodes:

```
match (n) return distinct labels(n);
```

// display's all labels existing in db
// distinct removes duplicate labels and node labels are case sensitive

Listing all labels for a node:

```
Match (n) where id(n)=0 return distinct labels(n)
```

//list all the labels associated with node with id=0

Database Management Systems

Neo4J – Remove/Update Labels to node (Contd..)

Count the labels available in db:

```
Match (n) return distinct count (labels(n));
//displays number of distinct labels in db
```

List all labels of all nodes:

```
Match (n) return distinct count (labels(n)),labels(n);
// returns label along with the label counts
```

Delete a node with a given label:

```
Match (n) where n:TeamLeader delete n
// deletes all the nodes with the label TeamLeader
```

Database Management Systems

Neo4J – Create a node with properties

Create a node with properties (Properties are attributes of a node)

Properties are always associated with a node or relationship

Properties are optional attributes which are represented as key-value pairs as ({})

A node can have zero or more properties

Creating a node with a single property:

create (x: Labelname {Property name: Value}) return x;

create (x: Book {title : “The Bengal Tiger”}) return x;

// creates a node with label Book and a property named title with its value “The Bengal Tiger”

Database Management Systems

Neo4J – Create a node with properties

Creating a node with multiple single properties:

Pair of properties are separated by comma

```
create (x: Labelname {Property_name: Value, Property_name : value}) return x;
```

```
create  (x: Book {title : “Database System”, author: “Navathe”, publisher : “Prearson”})
return x;
```

```
// creates a node with label Book with 3 properties.
//property names are case sensitive and can contain underscore and alpha numeric
characters
//must start with a letter
```

```
create  (x: Book1 {title : “Database System”, author: “Navathe”, Author : “Elmasri”})
return x;
// creates a node with label Book1 with 3 properties.
```

Database Management Systems

Neo4J – Data Types

To define properties of a node

Property is of the form **Key : Value** pair

Key is the name of the property – String

Value can be primitive data type or can be array of primitive datatypes

List of Primitive datatypes:

Boolean, integer, double, long, character, string

Array of primitive datatypes:

Boolean array, int array, double array, long array, char array, string array

// null is not a valid value for the property

Database Management Systems

Neo4J – Data Types

To define properties of a node

Property is of the form **Key : Value** pair

Key is the name of the property – String

Value can be primitive data type or can be array of primitive datatypes

List of Primitive datatypes:

Boolean, integer, double, long, character, string

Array of primitive datatypes:

Boolean array, int array, double array, long array, char array, string array

// null is not a valid value for the property

Neo4J – Data Types

To create a node with properties of different datatype:

```
Create (x:Book { title:"the  lives of others", author : "Neel", publisher: ["Chato & Windos","w norton"], price: 285.00, pages : 528, instock : false }) return x
```

// Creates a node with label Book and 6 properties, where title, author property is of type string, publisher of type string array, price is of type float, page is of type integer and instock is of type Boolean.

// creates a node with 6 properties as specified in query

```
Create (x:Book{details:["the nagas",172.00,398]}) return x;
```

// collections containing mixed types cannot be stored in properties, considers the datatype of first value

```
Create (x:Book{details:["the nagas","172.00","398"]}) return x;
```

//Successfull

Database Management Systems

Neo4J – Search nodes by using properties

To filter a node with properties and with specific value:

There are multiple nodes with label book and associated with properties like author, title, pages, price and publisher

Search all nodes where author of the book is “Chetan ”

Match (n:Label{property:”value”}) return n;

Match (n:Book{author:”Chetan”}) return n;

// return all the nodes with label book and property author with value “Chetan” , if not available returns zero records

Match (n:Book1{Author:"Elmasri"}) return n;
// returns nodes with label Book1 and Author value Elmasri

Database Management Systems

Neo4J – Search nodes by using properties

To filter a node using operators (AND, OR, < , >……)

Search all nodes whose author is a1 or a2 and price of the book is one thousand.

```
Match (n:Book) where n.price<1000 AND (n.author="Neel" or n.author =
"Navathe") return n;
// return all the nodes with label book with price <1000 and author with value
Neel or Navathe
```

Database Management Systems

Neo4J – Update properties of a Node

We created:

Create (x:Book { title:"the lives of others", author : "Neel", publisher: ["Chato & Windos","w norton"], price: 285.00, pages : 528, instock : false }) return x

To change the title of the Book

Match(n) where n.title="the lives of others**" set n.title="The lives" return n**

//updates the title from "the lives of others" to "the lives" and returns the node

Neo4J – Update properties of a Node

To update multiple properties:

```
Match(n:Book {title="existing title}) set
n.author="Gosh",n.price=324.00,n.stock=true return n
```

//Searches for the nodes with label Book and title ="existing title" and update its author, price and stock properties accordingly.

Neo4J – Delete properties of a Node

To delete multiple properties:

Match clause is used to search and remove clause to delete certain property.

Method 1: Remove property by setting the value to NULL

```
Match(n:Book {author:“Neel”}) set n.publisher=NULL return n
```

//to remove - set publisher to null from the nodes with label Book and author “Neel”
//This removes property publisher.

Method 2: Remove property by setting the value to NULL

```
Match(n:Book) where n.author = “Neel” set n.publisher=NULL return n
```

Database Management Systems

Neo4J – Delete properties of a Node

To delete property from all the nodes

Use Match clause to get all the nodes and remove clause to delete certain property from all.

Method 1: Remove property “pages” from all nodes

```
Match (n) remove n.pages return n
```

//to remove property pages from all the nodes

Method 2: Remove property “title” from the nodes with label bestseller

```
Match(n) where n:Bestseller remove n.title return n;
```

//a property can be used as a filter to delete other properties

Neo4J – Delete properties of a Node

To delete the nodes

Use match clause to select and delete clause to delete a node

```
Match (n) n.stock=false delete n
```

```
//to delete nodes based on the property stock with value “false”
// based if id the properties can be deleted
```

Remove a property named price from the node with the internal id = 1

```
Match(n) where id(n)=1 remove n.price return n
```

Database Management Systems

Neo4j Relationship

We can create a relationship using the CREATE clause. We will specify relationship within the square braces “[ ]” depending on the direction of the relationship it is placed between hyphen “ - ” and arrow “ → ” as shown in the following syntax.

Format : create (node1) – [:RelationshipType]->(node2)

Create two nodes and then relation ship between them

```
Create (chetan:author{name:”Chetan”,dob:1980,age:41})
Create (nagas:Book{title:”Nagas”,price:200,publication:”westland publication”}
//above to queries creates 2 nodes
Create (chetan)-[r:author_of]->(nagas)


//above query to create relationship between them
```

Database Management Systems

Neo4j Relationship

We can create a relationship using the CREATE clause. We will specify relationship within the square braces “[ ]” depending on the direction of the relationship it is placed between hyphen “ - ” and arrow “ → ” as shown in the following syntax.

Format : create (node1) – [:RelationshipType]->(node2)

Create two nodes and then relation ship between them

```
Create (chetan:author{name:”Chetan”,dob:1980,age:41})
Create (nagas:Book{title:”Nagas”,price:200,publication:”westland publication”}
//above to queries creates 2 nodes
Create (chetan)-[r:author_of]->(nagas)
//above query to create relationship between them
```

THANK YOU

Department of Computer Science and Engineering

Database Management Systems

In-Memory DataBase

Prof. Bhargavi M & Prof. Vidhyashree DM

Department of Computer Science and Engineering

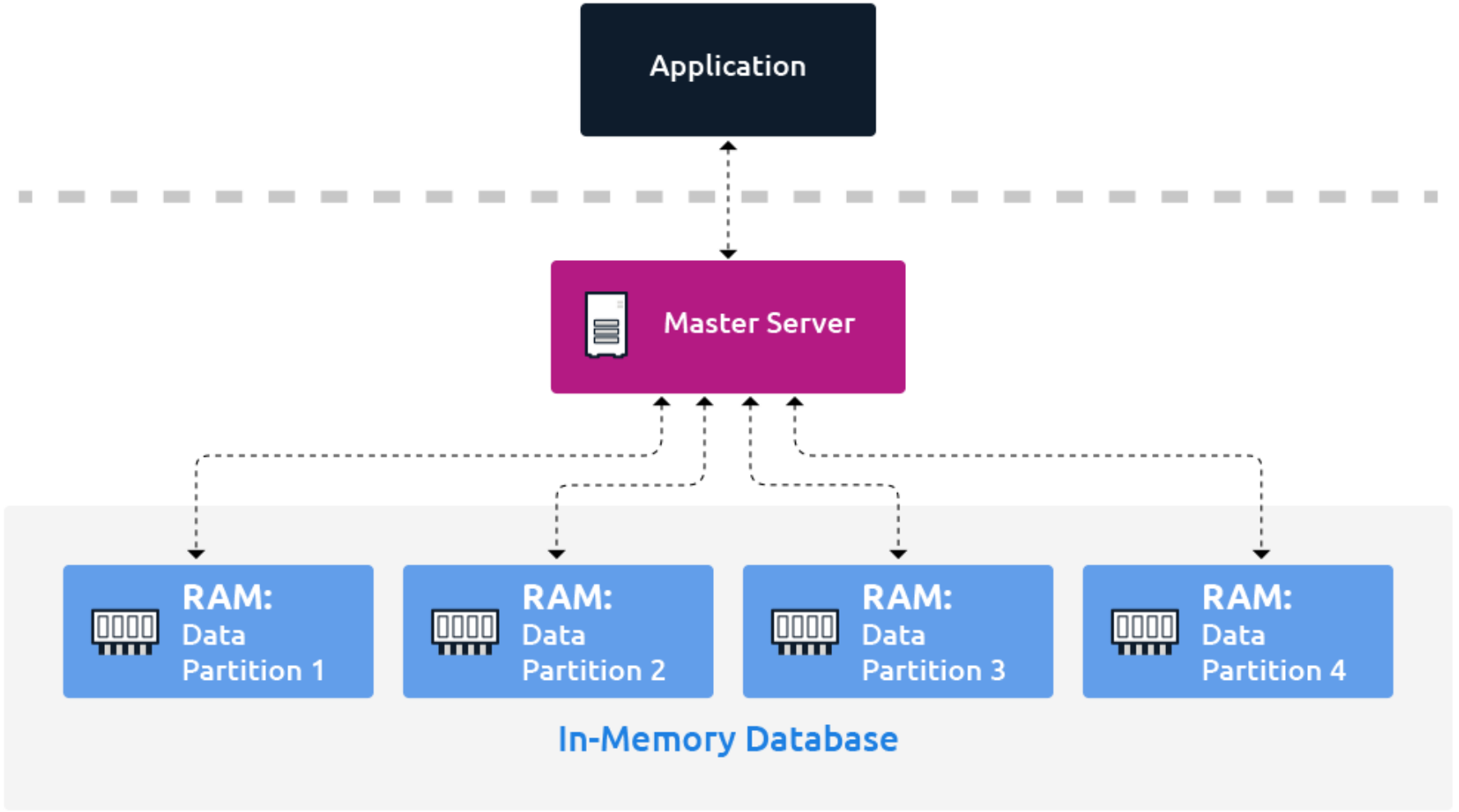
Database Management Systems

In-memory database

- An in-memory database is a type of purpose-built database that relies primarily on memory for data storage, in contrast to databases that store data on disk or SSDs.
- In-memory databases are designed to attain minimal response time by eliminating the need to access disks.
- Because all data is stored and managed exclusively in main memory, it is at risk of being lost upon a process or server failure.
- In-memory databases can persist data on disks by storing each operation in a log or by taking snapshots.
- In-memory databases are ideal for applications that require microsecond response times and can have large spikes in traffic coming at any time such as gaming leaderboards, session stores, and real-time analytics.

Database Management Systems

In-memory database



Database Management Systems

List of In-memory database

Apache Ignite

ArangoDB

Hazelcast

Infinity DB

Informix

SQLite

Microsoft SQL Server

OmniSci

VotlDB

Database Management Systems

In-memory database

Finance: Fraud Detection and Monitoring

Retail: Operations Efficiencies and Higher Sales Volumes

Healthcare: Clinical Risk and Patient Safety Monitoring

Network Threat Detection

Real time bidding

Gaming Leader boards

Caching

Database Management Systems

In-memory database - VoltDB

- VoltDB is a revolutionary new database product
- Best solution for high performance business-critical applications
- VoltDB focuses specifically on fast data.

Who Should use VoltDB???

Financial applications, social media applications, and field of Internet of Things.

VoltDB is used for capital markets data feeds, financial trade, telco record streams and sensor-based distribution systems.

It's also used in emerging applications like wireless, online gaming, fraud detection, digital ad exchanges and micro transaction systems

For more information refer:
https://docs.voltdb.com/UsingVoltDB/index.php

THANK YOU

Department of Computer Science and Engineering