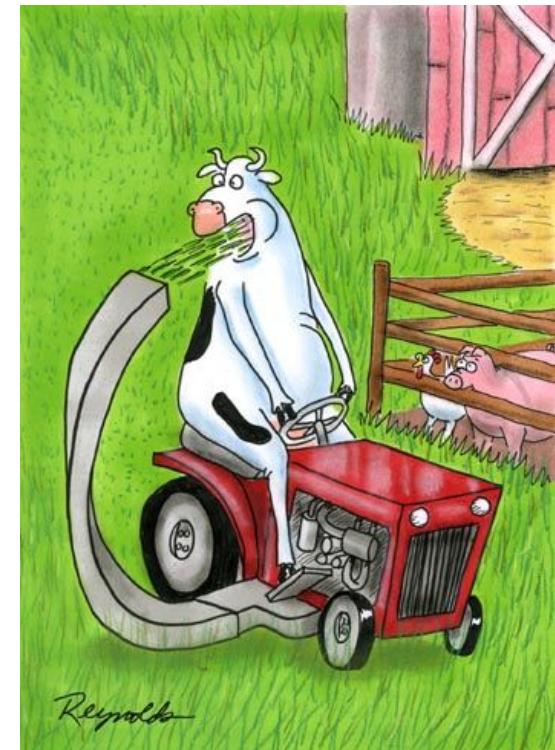




External Sorting

Problem Set #2 should be graded before Thursday

Problems Set #4 and Midterm results will be here when you get back from break





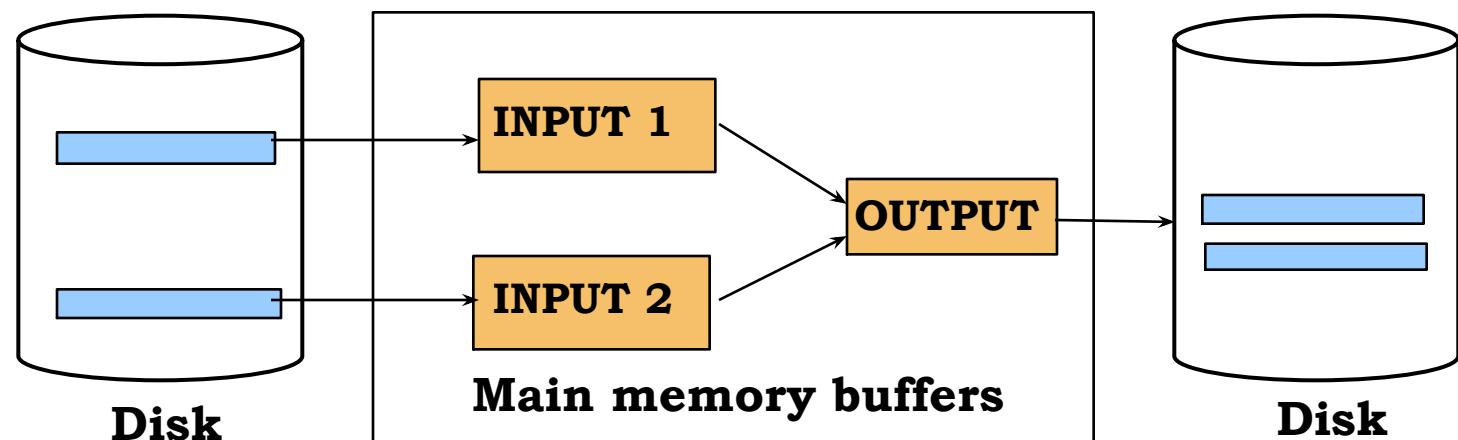
Why Sort?

- ❖ A classic problem in computer science!
- ❖ Advantages of requesting data in sorted order
 - gathers duplicates
 - allows for efficient searches
- ❖ Sorting is first step in *bulk loading* B+ tree index.
- ❖ *Sort-merge* join algorithm involves sorting.
- ❖ Problem: sort 20Gb of data with 1Gb of RAM.
 - why not let the OS handle it with virtual memory?



2-Way Sort: Requires 3 Buffers

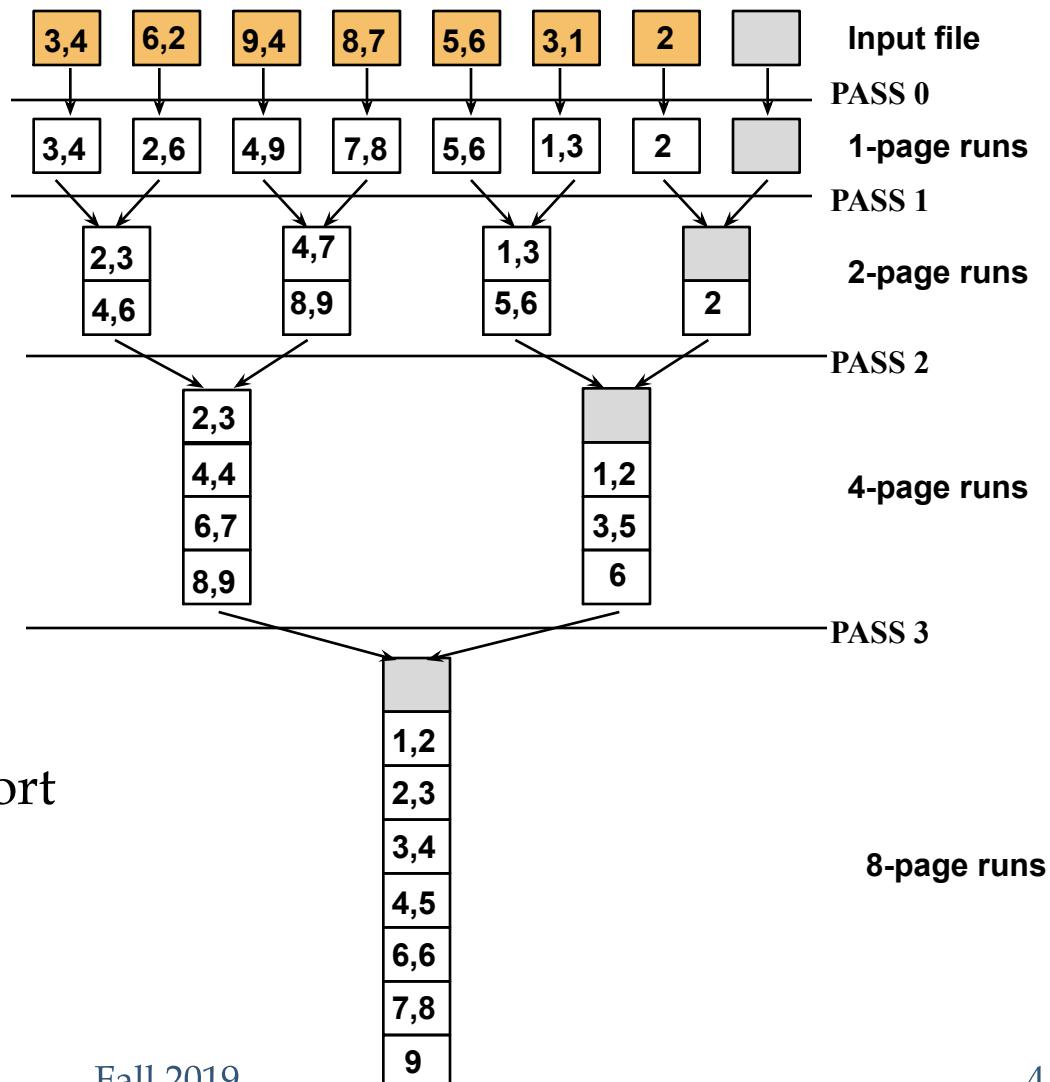
- ❖ Pass 1: Read a page, sort it, write it.
 - only one buffer page is used
- ❖ Pass 2, 3, ..., N etc.:
 - Read two pages, merge them, and write merged page
 - Requires three buffer pages.





Two-Way External Merge Sort

- ❖ Each pass we read + write each page in file.
- ❖ N pages in the file \Rightarrow the number of passes
 $= \lceil \log_2 N \rceil + 1$
- ❖ So total cost is
($2N = N$ reads + N writes):
$$2N(\lceil \log_2 N \rceil + 1)$$
- ❖ Idea: Divide and conquer: sort pages and merge





General External Merge Sort

☞ *More than 3 buffer pages. How can we utilize them?*

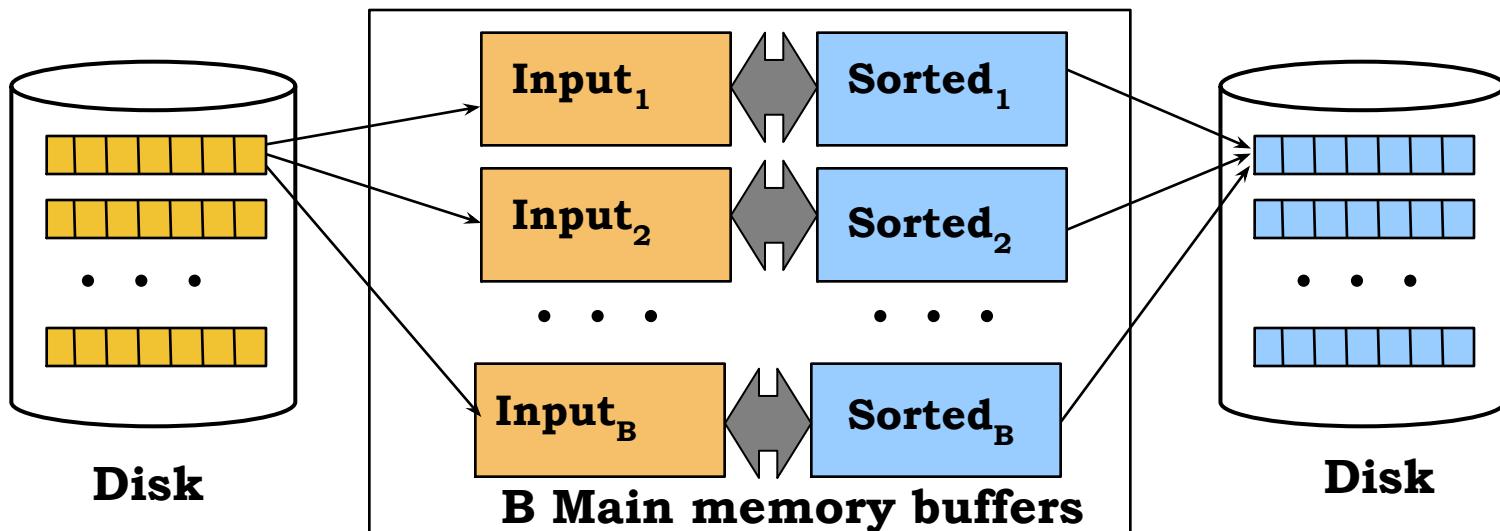
- ❖ Key Insight #1: We can merge more than 2 input buffers at a time... affects fanout \square base of log!
- ❖ Key Insight #2: The output buffer is generated incrementally, so only one buffer page is needed for any size of run!
- ❖ To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
 - Pass 2, ..., etc.: merge $B-1$ runs, leaving one page for output.



General External Merge Sort

☞ More than 3 buffer pages. How can we utilize them?

- ❖ To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.

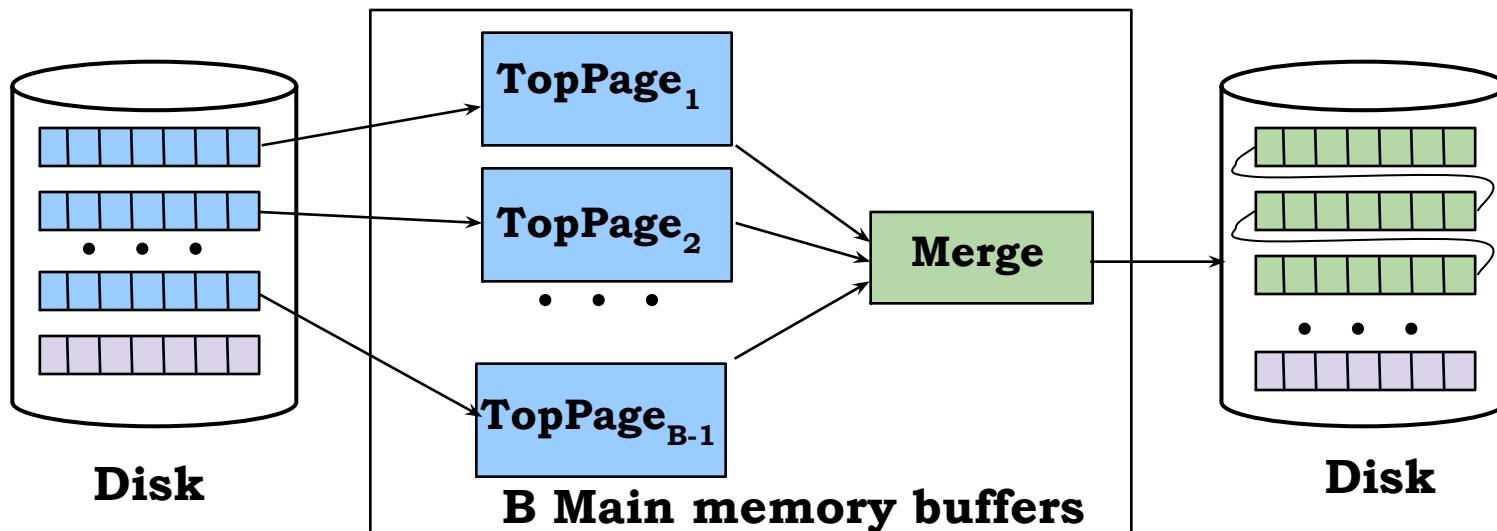




General External Merge Sort

☞ More than 3 buffer pages. How can we utilize them?

- ❖ To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
 - Pass 1, ..., etc.: merge $B-1$ runs. Repeat.





Cost of External Merge Sort

- ❖ Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- ❖ Cost = $2N * (\# \text{ of passes})$
- ❖ E.g., with 5 buffer pages, to sort 108 page file:
 - Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each
(last run is only 3 pages)
 - Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each
(last run is only 8 pages)
 - Pass 2: $\lceil 6 / 4 \rceil = 2$ sorted runs, 80 pages and 28 pages
 - Pass 3: Sorted file of 108 pages



Number External Sort Passes

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
1,000,000,000	30	15	10	8	5	4



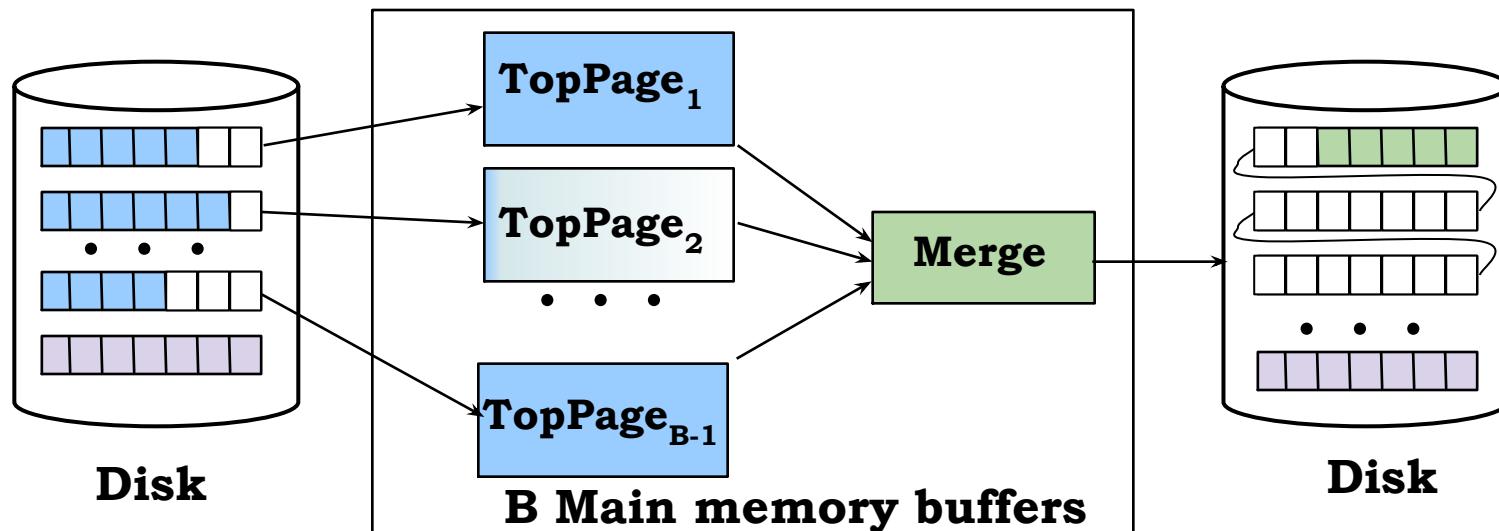
Internal Sort Algorithm

- ❖ Quicksort is a fast way to sort in memory.
 - Very fast on average
 - Worse case N^2 (i.e. bad pivots)
- ❖ Alternatives
 - Heap Sort, stable and always $O(N \log N)$
 - Merge Sort, same approach used in “out-of-core” sort but applied within a block recursively (low overhead)
 - Divides block into two halves, sorts each by dividing them recursively into two halves until there is only one item in the list. Then merges all of the "half-sized" lists while returning up the recursion.
- ❖ Another Problem... waiting to fill the buffer pool



Sorting Stalls

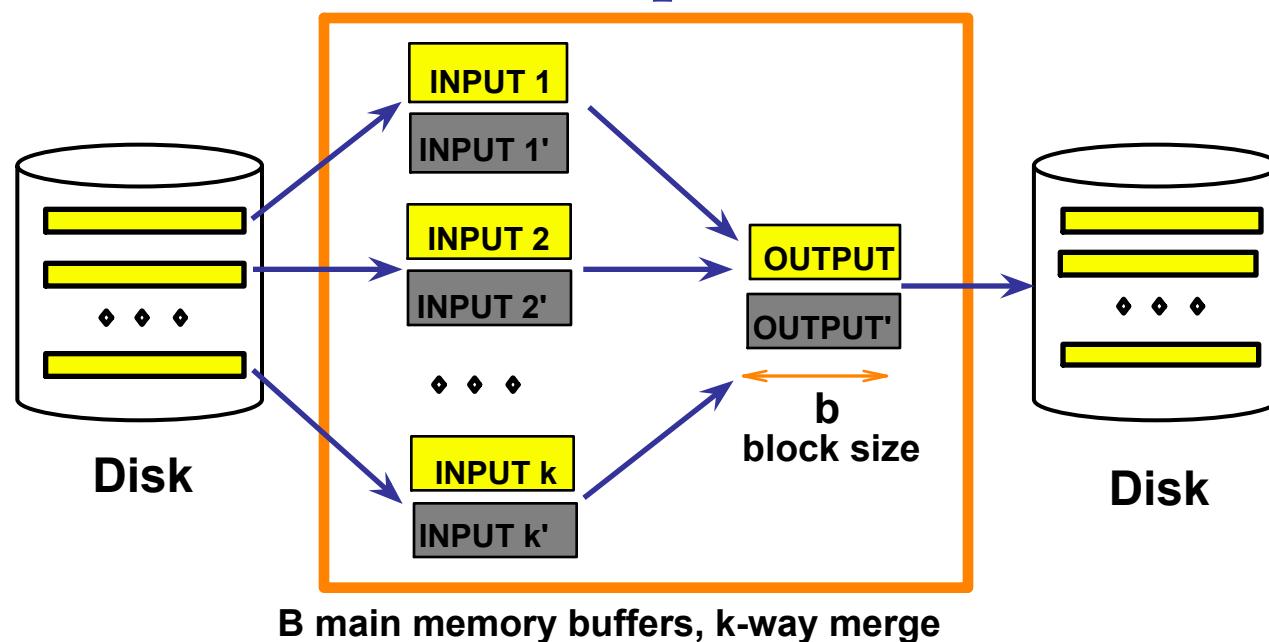
- ❖ When a "top page" empties, we need to wait for it to be refilled
- ❖ While waiting, we can't fill the merge output buffer using the other top pages, because the next value merged might come from the next block of the exhausted run





Double Buffering

- ❖ To reduce wait time for I/O request to complete, can *prefetch* into a "shadow block".
- ❖ Potentially, more passes; in practice, most files still sorted in 2-3 passes.





Sorting Records!

- ❖ Sorting has become a blood sport!
 - Parallel external sorting is the name of the game ...
- ❖ 2015 FuxiSort (Alibaba Group, Inc.)
 - Sort 100Tbyte of 100 byte records
 - Typical DBMS: > 10 days
 - World record: **329 seconds**
 - 2 Xeon E5-2630 2 2.3 GHz with 3134 nodes
 - Each node: 96 GB of RAM, and a 12x2 TB SATA disk
- ❖ New benchmarks proposed:
 - Minute Sort: How many can you sort in 1 minute?
 - Cloud Sort: How many \$ per TB sorted?



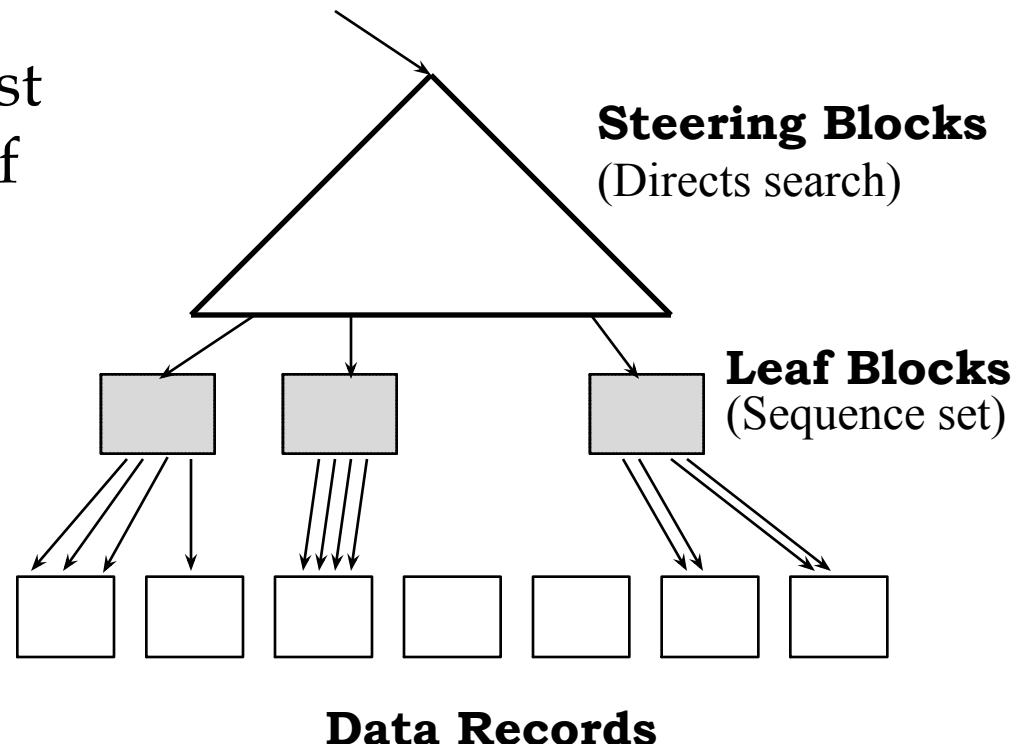
Using B+ Trees for Sorting

- ❖ Scenario: Table to be sorted has B+ tree index on sorting column(s).
- ❖ Idea: Can retrieve records in order by traversing leaf pages.
- ❖ *Is this a good idea?*
- ❖ Cases to consider:
 - B+ tree is clustered *Good idea!*
 - B+ tree is not clustered *Could be a very bad idea!*



Clustered B+ Tree Used for Sorting

- ❖ Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- ❖ If Alternative 2 is used?
Additional cost of retrieving data records:
each page fetched just once.
- ❖ Fill factor of < 100% introduces a small overhead extra pages fetched

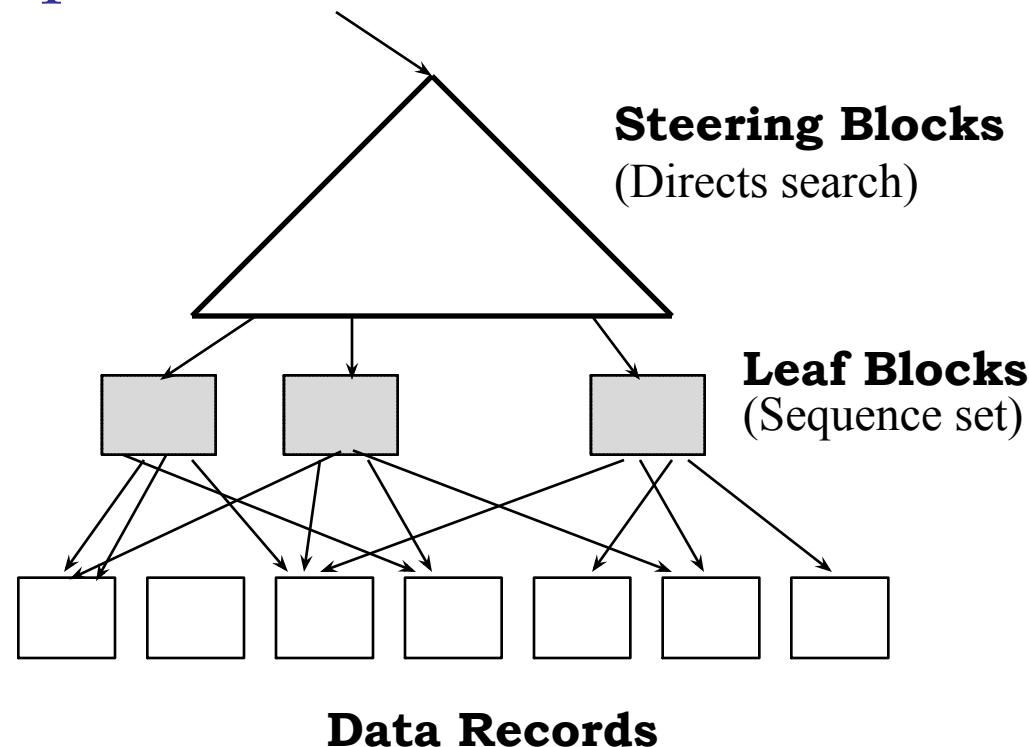


● *Always better than external sorting!*



Unclustered B+ Tree Used for Sorting

- ❖ Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!





External Sorting vs. Unclustered Index

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

- p : # of records per page
- $B=1,000$ and block size=32 for sorting
- $p=100$ is the more realistic value.



Summary

- ❖ External sorting is important; DBMS may dedicate part of buffer pool just for sorting!
- ❖ External merge sort minimizes disk I/O cost:
 - Pass 0: Produces sorted *runs* of size B (# buffer pages).
Later passes: *merge* runs.
 - # of runs merged at a time depends on B , and *block size*.
 - Larger block size means less I/O cost per page.
 - Larger block size means smaller # runs merged.
 - In practice, # of runs rarely more than 2 or 3.



Summary, cont.

- ❖ Choice of internal sort algorithm may matter:
 - Quicksort: Quick!
 - Alternative sorts
 - guaranteed $N \log N$ on worst case data
 - stable (ties retain their original order)
- ❖ The best sorts are wildly fast:
 - Despite 40+ years of research, we're still improving!
- ❖ Clustered B+ tree is good for sorting; unclustered tree is usually very bad.



Next Time

Schema Refinement



...have a good break!



Schema Refinement and Normal Forms

PS2 graded, Midterm results on Thursday 





Back to database design

- ❖ Which tables to include?
 - Are all choices equally good?
 - Were there other choices?
 - Are some choices better than others
- ❖ Which attributes in which tables?

Roster										
player	height	weight	college	dob	team	year	position	jersey	games	starts
Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Radiers	1991	LB	52	16	0

- ❖ What distinguishes a "good" database design from a "bad" one"



The Evils of Redundancy

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



Functional Dependencies (FDs)

- ❖ A functional dependency $X \square Y$ holds over relation R if, for every allowable instance r of R:
 - for all $t_1 \in r, t_2 \in r, \pi_X(t_1) = \pi_X(t_2)$ implies $\pi_Y(t_1) = \pi_Y(t_2)$
 - i.e., given two tuples in r , if the X values match, then the Y values must also match. (X and Y are *sets* of attributes.)
- ❖ An FD is a statement about *all* allowable relations.
 - Must be identified based on semantics of application.
 - Given some allowable instance r_1 of R, we can check if it violates some FD f , but we cannot tell if f holds over R!
- ❖ K is a candidate key for R means that $K \square R$
 - However, $K \square R$ does not require K to be *minimal*!



Example: Constraints on Entity Set

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



Example (Contd.)

❖ Problems due to R \square W :

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

Hourly_Emps

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

Hourly_Emps2

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

Wages

Will 2 smaller tables be better?



Reasoning About FDs

- ❖ Given some FDs, we can usually infer additional FDs:
 - $\text{zip} \sqsubset \text{state}$, $\text{state} \sqsubset \text{senator}$ implies $\text{zip} \sqsubset \text{senator}$
- ❖ An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.
 - $F^+ = \text{closure of } F$ is the **set of all FDs** that are implied by F .
- ❖ Armstrong's Axioms (X, Y, Z are sets of attributes):
 - **Reflexivity:** If $X \subseteq Y$, then $Y \sqsubset X$ ($\text{city}, \text{state}, \text{zip} \sqsubset \text{city}, \text{state}$)
 - **Augmentation:** If $X \sqsubset Y$, then $XZ \sqsubset YZ$ for any Z
($\text{city}, \text{state} \sqsubset \text{zip}$, then $\text{addr}, \text{city}, \text{state} \sqsubset \text{addr}, \text{zip}$)
 - **Transitivity:** If $X \sqsubset Y$ and $Y \sqsubset Z$, then $X \sqsubset Z$
- ❖ These are **sound** and **complete** inference rules for FDs!
 - **sound:** they will generate only FDs in F^+
 - **complete:** repeated applications will generate all FDs in F^+



Reasoning About FDs (Contd.)

- ❖ Couple of additional rules (that follow from AA):
 - *Union*: If $X \sqsubseteq Y$ and $X \sqsubseteq Z$, then $X \sqsubseteq YZ$
 - *Decomposition*: If $X \sqsubseteq YZ$, then $X \sqsubseteq Y$ and $X \sqsubseteq Z$
- ❖ Example: Contracts(*cid*,*sid,jid,did,pid,qty,value*), and:
 - C is the key: $C \sqsubseteq CSJDPQV$
 - Projects purchase each part using single contract: $JP \sqsubseteq C$
 - Dept purchase at most one part from a supplier: $SD \sqsubseteq P$
- ❖ $JP \sqsubseteq C$, $C \sqsubseteq CSJDPQV$ imply $JP \sqsubseteq CSJDPQV$
- ❖ $SD \sqsubseteq P$ implies $SDJ \sqsubseteq JP$
- ❖ $SDJ \sqsubseteq JP$, $JP \sqsubseteq CSJDPQV$ imply $SDJ \sqsubseteq CSJDPQV$



Reasoning About FDs (Contd.)

- ❖ Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- ❖ Typically, we just want to check if a given FD $X \square Y$ is in the closure of a set of FDs F . An efficient check:
 - Compute attribute closure of X (denoted X^+) wrt F :
 - Set of all attributes A such that $X \square A$ is in F^+
 - There is a linear time algorithm to compute this.
 - Check if Y is in X^+



Algorithm for test if FD is in F^+

- ❖ Given $X \square Y$

closure = X;

repeat {

if there is an FD $U \square V$ in F such that $U \subseteq closure:$

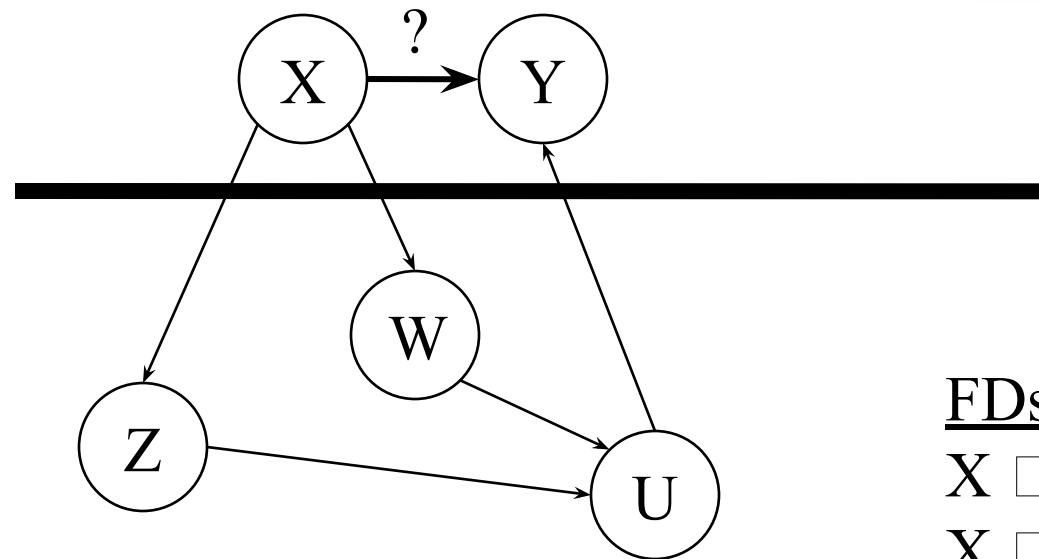
closure = closure \cup V

} until closure does not change

- ❖ Consider $X \square Y$ as a graph with X and Y as nodes and a directed edge from X to Y .
- ❖ Traverse the set of *given* FDs to extend all existing paths
- ❖ When the path can be extended no farther determine if there is a path from $X \square Y$



Example Check



FDs:

$X \sqsubseteq W$

$X \sqsubseteq Z$

$WZ \sqsubseteq U$

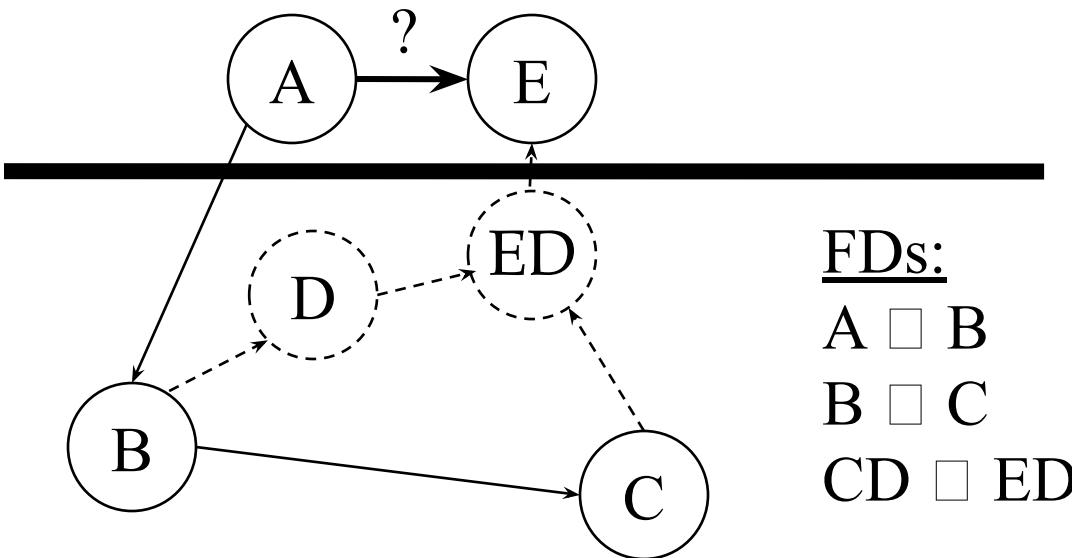
$U \sqsubseteq Y$

- ❖ Does $F = \{A \sqsubseteq B, B \sqsubseteq C, CD \sqsubseteq ED\}$ imply $A \sqsubseteq E$?
 - i.e., is $A \sqsubseteq E$ in the closure F^+ ? Equivalently, is E in A^+ ?



Try Again

- Does $F = \{A \sqsubset B, B \sqsubset C, CD \sqsubset ED\}$ imply $A \sqsubset E$?
 - i.e, is $A \sqsubset E$ in the closure F^+ ? Equivalently, is E in A^+ ?



Since D is not in our set when we attempt to add $CD \rightarrow ED$ our algorithm terminates. In the resulting graph there is no edge from A to E .

If $B \rightarrow D$ then a path would be created.



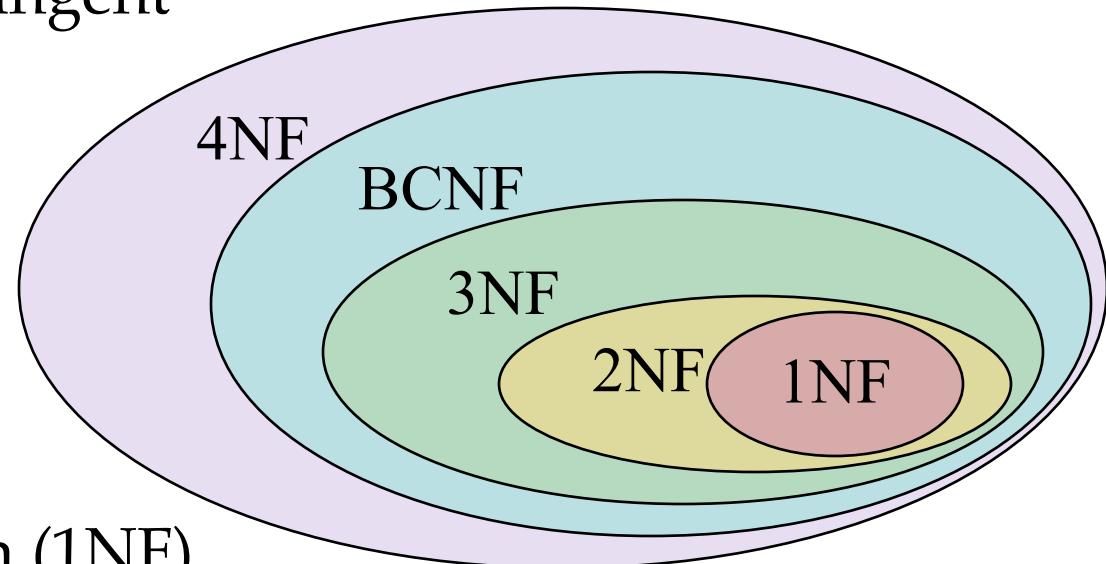
Normal Forms

- ❖ To eliminate redundancy and potential update anomalies, one can identify generic templates called “normal forms”
- ❖ If a relation is in a certain *normal form* (Boyce-Codd Normal Form (BCNF), third normal form (3NF) etc.), it is known that certain kinds of redundancy are avoided/minimized.
- ❖ This can be used to help us decide whether decomposing the relation will help.
- ❖ Role of FDs in detecting redundancy:
 - Consider a relation R with 3 attributes, ABC.
 - No FDs hold: There is no redundancy here.
 - Given $A \sqsubset\!\!\! \sqsubset B$: Several tuples could have the same A value, and if so, they'll all have the same B value!



Normal Form Hierarchy

- ❖ An increasingly stringent hierarchy of “Normal Forms”
- ❖ Each outer form trivially satisfies the requirements of inner forms
- ❖ The 1st normal form (1NF) is part of the definition of the relational model. Relations must be sets (unique) and all attributes atomic (not multiple fields or variable length records).
- ❖ The 2nd normal form (2NF) requires schemas not have any FD, $X \square Y$, where X as a strict subset of the schema's key.





Boyce-Codd Normal Form (BCNF)

- ❖ Relation R with FDs F is in BCNF if, for all $X \sqsupseteq A$ in F^+
 - $A \in X$ (called a *trivial* FD), or 
 - X contains a key for R.
- ❖ In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
- ❖ BCNF considers *all domain keys*, not just the *primary* one
- ❖ BCNF schemas do not contain redundant information that arise from FDs

Includes silly FDs like:
(city, state) \sqsupseteq state



BCNF Examples

- ❖ In BCNF

Person(First, Last, Address, Phone)

Functional Dependencies: $FL \rightarrow A$, $FL \rightarrow P$

- ❖ Not in BCNF

Person(First, Last, Address, Phone, Email)

An attempt to allow a person to have
multiple emails.

Functional Dependencies: $FL \rightarrow A$, $FL \rightarrow P$



Third Normal Form (3NF)

- ❖ Reln R with FDs F is in 3NF if, for all $X \sqsupseteq A$ in F^+
 - $A \in X$ (called a *trivial* FD), or
 - X contains a key for R, or
 - A is part of some key for R.
- ❖ *Minimality* of a key is crucial in third condition above!
- ❖ If R is in BCNF, it is trivially in 3NF.
- ❖ If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no “good” decomp, or performance considerations).



3NF Examples

- ❖ Phonebook where friends have multiple addresses
- ❖ In 3NF, not in BCNF

Person(First, Last, Addr, Phone)

Functional Dependencies:

FLA \square P, P \square A

- ❖ Not in 3NF or BCNF

Person(First, Last, Addr, Phone, Mobile)

Functional Dependencies:

FLA \square P, P \square A, FL \square M



What Does 3NF Achieve?

- ❖ If 3NF is violated by $X \rightarrow A$, one of the following holds:
 - X is a proper subset of some key K (partial dependency)
 - We store (X, A) pairs redundantly.
 - X is not a proper subset of any key (transitive dependency).
 - There is a chain of FDs $K \rightarrow X \rightarrow A$, which means that we cannot associate an X value with a K value unless we also associate an A value with an X value.
- ❖ But, even if relation is in 3NF, problems can arise.



Lingering 3NF Redundancies

- ❖ Revisiting an old Schema

Reserves(Sailor, Boat, Date, CreditCardNo)

FDs: SBD ⊓ SBDC, C ⊓ S

- ❖ In 3NF, but database likely stores many redundant copies of the (C, S) tuple
- ❖ Thus, 3NF is indeed a compromise relative to BCNF.



Decomposition of a Relation Scheme

- ❖ Suppose that relation R contains attributes $A_1 \dots A_n$.
A decomposition of R consists of replacing R by two or more relations such that:
 - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
 - Every attribute of R appears as an attribute of one of the new relations.
- ❖ Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- ❖ E.g., Can decompose **SNLRWH** into **SNLRH** and **RW**.



Example Decomposition

- ❖ Decompositions should be used only when needed.
 - SNLRWH has FDs $S \rightarrow\!\!\!-\! R$ and $R \rightarrow\!\!\!-\! W$
 - Second FD violates 3NF
(R is not a key, W is not part of a key)
 - Redundancy: W values repeatedly associated with R values.
 - Easiest fix; create a relation RW to store these associations, and to remove W from the main schema:
 - i.e., we decompose SNLRWH into SNLRH and RW
- ❖ Given SNLRWH tuples, we just store the projections SNLRH and RW, are there any potential problems that we should be aware of?



Problems with Decompositions

❖ There are three potential problems to consider:

Problem 1) Some queries become more expensive.

- e.g., How much did Joe earn? ($\text{salary} = \text{W}^*\text{H}$)

Problem 2) Given instances of the decomposed relations, we may not be able to reconstruct the corresponding original relation!

- Fortunately, not in the SNLRWH example.

Problem 3) Checking some dependencies may require joining the instances of the decomposed relations.

- Fortunately, not in the SNLRWH example.

❖ Tradeoff: Must consider these issues vs. redundancy.



Lossless Join Decompositions

- ❖ Decomposition of R into X and Y is lossless-join w.r.t. a set of FDs F if, for every instance r that satisfies F:
 $ABCD = (\text{SELECT } B,C \text{ FROM } ABCD) \text{ NATURAL JOIN } (\text{SELECT } A,B,D \text{ FROM } ABCD)$
- ❖ It is always true that $ABCD \subseteq BC \text{ NATURAL JOIN } ABD$
 - In general, the other direction does not hold!
If equal, the decomposition is lossless-join.
- ❖ Definition extended to decomposition into 3 or more relations in a straightforward way.
- ❖ *It is essential that all decompositions used to eliminate redundancy be lossless! (Avoids Problem 2)*

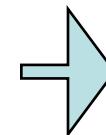


More on Lossless Join

- The decomposition of R into X and Y is **lossless-join wrt F** if and only if the closure of F contains:
 - $X \cap Y \sqsubseteq X$, or
 - $X \cap Y \sqsubseteq Y$(in other words the attributes common to X and Y must contain a key for either X or Y)
- In particular, the decomposition of R into UV and R - V is lossless-join if $U \sqsubseteq V$ holds over R.

$$ABC \Rightarrow AB, BC$$

A	B	C
1	2	3
4	5	6
7	2	8



A	B
1	2
4	5
7	2

JOIN

B	C
2	3
5	6
2	8

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3



Not lossless

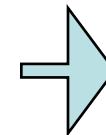


More on Lossless Join

- The decomposition of R into X and Y is **lossless-join wrt F** if and only if the closure of F contains:
 - $X \cap Y \sqsubseteq X$, or
 - $X \cap Y \sqsubseteq Y$(in other words the attributes common to X and Y must contain a key for either X or Y)
- In particular, the decomposition of R into UV and R - V is lossless-join if $U \sqsubseteq V$ holds over R.

$$ABC \Rightarrow AB, AC$$

A	B	C
1	2	3
4	5	6
7	2	8



A	B
1	2
4	5
7	2

JOIN

A	C
1	3
4	6
7	8

A	B	C
1	2	3
4	5	6
7	2	8



Lossless



Dependency Preserving Decomposition

Contracts(Cid,Sid,Jid,Did,Pid,Qty,Value)

- ❖ Consider CSJDPQV, C is key, JP ⊑ C and SD ⊑ P.
 - BCNF decomposition: CSJDQV and SDP
 - Problem: Checking JP ⊑ C requires a join!
- ❖ Dependency preserving decomposition (Intuitive):
 - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold.
- ❖ *Projection of set of FDs F:* If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs $U \sqsubseteq V$ in F^+ (*closure of F*) such that U, V are in X.



Dependency Preserving Decomposition

- ❖ Decomposition of R into X and Y is dependency preserving
if $(F_X \cup F_Y)^+ = F^+$
 - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+ .
- ❖ MUST consider F^+ , (not just F), in this definition:
 - ABC, $A \sqsubset B$, $B \sqsubset C$, $C \sqsubset A$, decomposed into AB and BC.
 - Is this dependency preserving? Is $C \sqsubset A$ preserved?????
- ❖ Dependency preserving does not imply lossless join:
 - ABC, $A \sqsubset B$, decomposed into AB and BC.
- ❖ And vice-versa!



Decomposition into BCNF

- ❖ Consider relation R with FDs F.
If $X \sqsubset\!\!\! \sqsubset Y$ violates BCNF,
decompose R into $R - Y$ and \underline{XY} .
 - Repeated applications of this rule gives relations in BCNF;
lossless join decomposition, and is guaranteed to terminate.
- ❖ Example: CSJDPQV, $SD \sqsubset\!\!\! \sqsubset P$, $J \sqsubset\!\!\! \sqsubset S$ (new),
(ignoring $JP \sqsubset\!\!\! \sqsubset C$ for now)
 - To deal with $SD \sqsubset\!\!\! \sqsubset P$, decompose into SDP, CSJDQV.
 - To deal with $J \sqsubset\!\!\! \sqsubset S$, decompose CSJDQV into JS and CJDQV
- ❖ The order in which we process violations can lead to a different set of relations!



BCNF and Dependency Preservation

- ❖ In general, there may not be a dependency preserving decomposition into BCNF.
 - e.g., CSZ, CS \sqsubseteq Z, Z \sqsubseteq C
 - Can't decompose while preserving 1st FD; not in BCNF.
- ❖ Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs: $\text{JP} \sqsubseteq \text{C}$, $\text{SD} \sqsubseteq \text{P}$ and $\text{J} \sqsubseteq \text{S}$).
 - However, it is a lossless join decomposition.
 - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
 - JPC tuples stored *only* for checking FD! (*Adds Redundancy!*)



Decomposition into 3NF

- ❖ Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, it can stop earlier).
- ❖ To ensure dependency preservation, one idea:
 - If $X \sqsubset Y$ is not preserved, add relation XY.
 - Problem is that XY may violate 3NF! e.g., consider the addition of CJP to “preserve” $JP \sqsubset C$. What if we also have $J \sqsubset C$?
- ❖ Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.



Minimal Cover for a Set of FDs

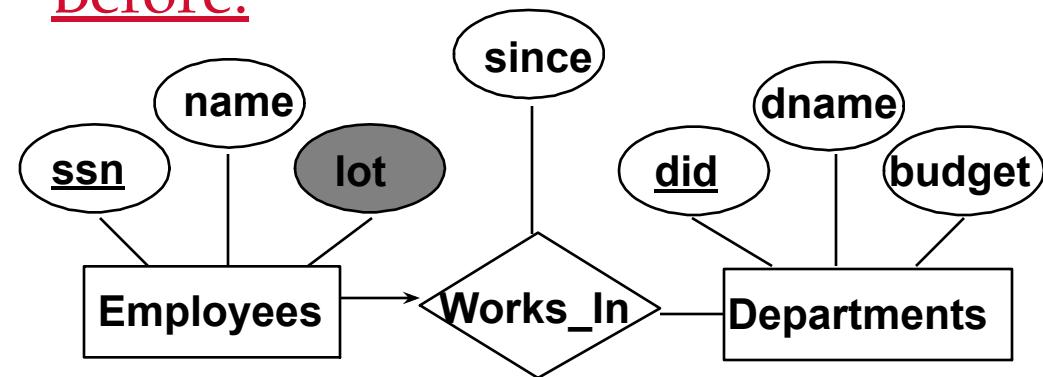
- ❖ Properties of a Minimal cover, G, for a set of FDs F:
 - Closure of F = closure of G.
 - Right hand side of each FD in G is a single attribute.
 - If we modify G by deleting a FD or by deleting attributes from an FD in G, the closure changes.
- ❖ Intuitively, every FD in G is needed, and is “*as small as possible*” in order to get the same closure as F.
- ❖ e.g., A \square B, ABCD \square E, EF \square GH, ACDF \square EG has the following minimal cover:
 - A \square B, ACD \square E, EF \square G and EF \square H
- ❖ M.C. → Lossless-Join, Dep. Pres. Decomp!!!



Refining an Entities and Relations

- ❖ 1st diagram translated:
 $\text{EmpWorksIn}(\underline{S}, \underline{N}, \underline{L}, \underline{D}, \underline{C})$
 $\text{Dept}(\underline{D}, \underline{M}, \underline{B})$
 - Lots associated with workers.
- ❖ Suppose all workers in a dept are assigned the same lot: $D \sqsubset L$
- ❖ And Employees start their new lot at a given date: $SL \sqsubset C$
- ❖ Redundancy is fixed by:
 $\text{Emp}(\underline{S}, \underline{N})$
 $\text{WorksIn}(\underline{S}, \underline{L}, \underline{D}, \underline{C})$ (note 3NF)
 $\text{Dept}(\underline{D}, \underline{M}, \underline{B})$
- ❖ Enforcement of FD: $D \sqsubset L$ is supported by
 $\text{DeptLot}(\underline{D}, \underline{L})$

Before:





Normalization example

Consider our NFL Roster table

Roster												
id	player	height	weight	college	dob	team	year	position	jersey	games	starts	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Radiers	1991	LB	52	16	0	

Initial list of functional dependancies:

$I \rightarrow PHWCD$

$PCD \rightarrow I$

$ITY \rightarrow PJGS$

$JTY \rightarrow I$



1st Normal Form (1NF)

A relation is 1NF if *all rows are distinct* and *all columns are single-valued*.
A typical unnormalized table is shown below. Notice how repeated items are expanded

Roster Unnormalized												
id	name	height	weight	college	dob	team	year	position	jersey	games	starts	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Raiders	1991	LB	52	16	0	
						Los Angeles Raiders	1992	LB	52	16	0	
						Los Angeles Raiders	1993	LB	52	16	2	
						Los Angeles Raiders	1994	LB	52	16	1	
						Oakland Raiders	1995	LB	52	16	16	
						Oakland Raiders	1996	LB	52	15	15	
						St Louis Rams	1997	LB	52	16	16	
						St Louis Rams	1998	LB	52	16	16	
						St Louis Rams	1999	LB	52	16	16	
						St Louis Rams	2000	LB	52	16	16	
						Pittsburgh Steelers	2001	LB	51	15	0	
						Pittsburgh Steelers	2002	LB	95	6	1	
						Oakland Raiders	2002	LB	52	3	0	
20001	Mike Jones	5-11	181	NC State	???	Pheonix Cardinals	1991	DT	96	11	3	



1st Normal Form (1NF)

The following "normalized" version of is in 1NF, but now there is considerable redundancy...

Roster												
id	name	height	weight	college	dob	team	year	position	jersey	games	starts	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Raiders	1991	LB	52	16	0	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Raiders	1992	LB	52	16	0	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Raiders	1993	LB	52	16	2	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Los Angeles Raiders	1994	LB	52	16	1	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Oakland Raiders	1995	LB	52	16	16	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Oakland Raiders	1996	LB	52	15	15	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	St Louis Rams	1997	LB	52	16	16	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	St Louis Rams	1998	LB	52	16	16	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	St Louis Rams	1999	LB	52	16	16	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	St Louis Rams	2000	LB	52	16	16	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Pittsburgh Steelers	2001	LB	51	15	0	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Pittsburgh Steelers	2002	LB	95	6	1	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	Oakland Raiders	2002	LB	52	3	0	
20001	Mike Jones	5-11	181	NC State	null	Phoenix Cardinals	1991	DT	96	11	3	



1st Normal Form (1NF)

A decomposition of Roster into two 1NF tables that eliminates redundancy in the same spirit as the original unnormalized table.

FD: I → NHWCD
PCD → I
ITY → PJGS
JTY → I

There is still lots of redundancy...

Player						
<u>id</u>	name	height	weight	college	dob	
20000	Mike Jones	6-1	240	Missouri	1969-04-15	
20001	Mike Jones	5-11	181	NC State	???	
PlayedFor						
<u>id</u>	<u>team</u>	<u>year</u>	position	jersey	games	starts
20000	Los Angeles Raiders	1991	LB	52	16	0
20000	Los Angeles Raiders	1992	LB	52	16	0
20000	Los Angeles Raiders	1993	LB	52	16	2
20000	Los Angeles Raiders	1994	LB	52	16	1
20000	Oakland Raiders	1995	LB	52	16	16
20000	Oakland Raiders	1996	LB	52	15	15
20000	St Louis Rams	1997	LB	52	16	16
20000	St Louis Rams	1998	LB	52	16	16
20000	St Louis Rams	1999	LB	52	16	16
20000	St Louis Rams	2000	LB	52	16	16
20000	Pittsburgh Steelers	2001	LB	51	15	0
20000	Pittsburgh Steelers	2002	LB	95	6	1
20000	Oakland Raiders	2002	LB	52	3	0
20001	Pheonix Cardinals	1991	DT	96	11	3



Discovery of new FDs

At this point we notice the the notion of a "team" and a "team's location" are not well represented in our table, so we split the team column into two.

FD: $I \rightarrow NHWCD$
 $PCD \rightarrow I$
 $ITY \rightarrow LPJGS$
 $JTY \rightarrow I$
 $TY \rightarrow L$

PlayedFor								
<u><i>id</i></u>	location	<u><i>team</i></u>	<u><i>year</i></u>	position	jersey	games	starts	
20000	Los Angeles	Raiders	1991	LB	52	16	0	
20000	Los Angeles	Raiders	1992	LB	52	16	0	
20000	Los Angeles	Raiders	1993	LB	52	16	2	
20000	Los Angeles	Raiders	1994	LB	52	16	1	
20000	Oakland	Raiders	1995	LB	52	16	16	
20000	Oakland	Raiders	1996	LB	52	15	15	
20000	St Louis	Rams	1997	LB	52	16	16	
20000	St Louis	Rams	1998	LB	52	16	16	
20000	St Louis	Rams	1999	LB	52	16	16	
20000	St Louis	Rams	2000	LB	52	16	16	
20000	Pittsburgh	Steelers	2001	LB	51	15	0	
20000	Pittsburgh	Steelers	2002	LB	95	6	1	
20000	Oakland	Raiders	2002	LB	52	3	0	
20001	Pheonix	Cardinals	1991	DT	96	11	3	



Discovery of new FDs

Our notion of a "team" is still less than ideal since a scan through the table exposes that 1) mascots of teams have changed and 2) old mascots have been reused by later teams, 3) But, in no year did two teams have the same mascot.

We remedy this by
adding a *tid* number,
which allows mascots
of teams to change

I → NHWCD

PCD → I

ITY → LPJGS

JTY → I

TY→ LM

MY→ T

PlayedFor								
<i>id</i>	location	<i>tid</i>	mascot	<u>year</u>	position	jersey	games	starts
20000	Los Angeles	1024	Raiders	1991	LB	52	16	0
...								
20000	Oakland	1024	Raiders	1995	LB	52	16	16
20000	Oakland	1024	Raiders	1996	LB	52	15	15
20000	St Louis	1025	Rams	1997	LB	52	16	16
...								
20000	Pittsburgh	1030	Steelers	2001	LB	51	15	0
20000	Pittsburgh	1030	Steelers	2002	LB	95	6	1
20000	Oakland	1024	Raiders	2002	LB	52	3	0
20001	Phoenix	1007	Cardinals	1991	DT	96	11	3



2nd Normal Form

A relation is in 2nd Normal Form if it is in 1NF and *every non-primary-key attribute of a table is fully functionally dependent on the primary key.*

In other words there are no partial dependances. A partial dependency is when a subset of attributes depend only upon a *subset* of the attributes of the table's primary key.

I → PHWCD

PCD → I

ITY → LMPJGS

TY → LM

MY → T

JTY → I

Player					
<u>id</u>	name	height	weight	college	dob
20000	Mike Jones	6-1	240	Missouri	1969-04-15
20001	Mike Jones	5-11	181	NC State	???

PlayedFor								
<u>id</u>	location	<u>tid</u>	mascot	<u>year</u>	position	jersey	games	starts
20000	Los Angeles	1024	Raiders	1991	LB	52	16	0
...								
20000	Oakland	1024	Raiders	1995	LB	52	16	16
20000	Oakland	1024	Raiders	1996	LB	52	15	15
20000	St Louis	1025	Rams	1997	LB	52	16	16
...								
20000	Pittsburgh	1030	Steelers	2001	LB	51	15	0
20000	Pittsburgh	1030	Steelers	2002	LB	95	6	1
20000	Oakland	1024	Raiders	2002	LB	52	3	0
20001	Pheonix	1007	Cardinals	1991	DT	96	11	3



2nd Normal Form

At this point we can decompose PlayedFor into two tables all of which are in 2NF. What redundancies are eliminated? What redundancies remain?

$I \rightarrow PHWCD, PCD \rightarrow I$

$\underline{ITY} \rightarrow PJGS, JTY \rightarrow I$

$\underline{TY} \rightarrow LM, MY \rightarrow L$

Team			
<u>tid</u>	<u>year</u>	location	mascot
1024	1991	Los Angeles	Raiders
...			
1024	1995	Oakland	Raiders
1024	1996	Oakland	Raiders
1025	1997	St Louis	Rams
...			
1030	2001	Pittsburgh	Steelers
1030	2002	Pittsburgh	Steelers
1024	2002	Oakland	Raiders
1007	1991	Pheonix	Cardinals

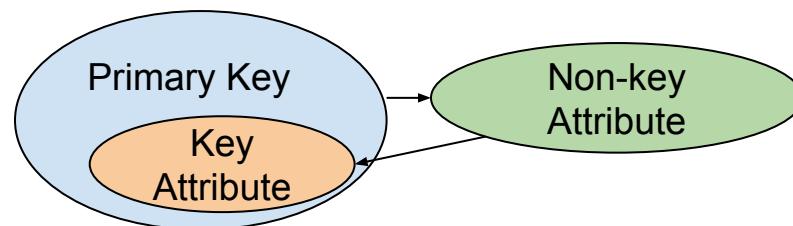
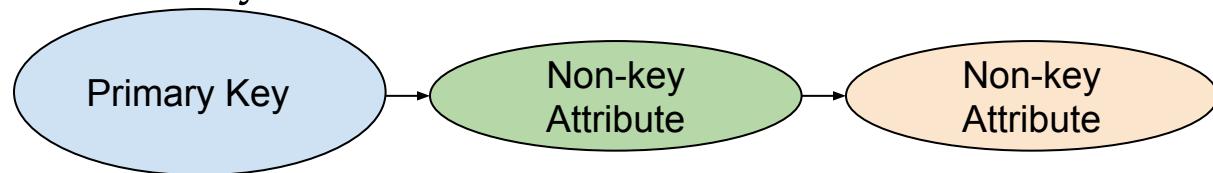
Player					
<u>id</u>	name	height	weight	college	dob
20000	Mike Jones	6-1	240	Missouri	1969-04-15
20001	Mike Jones	5-11	181	NC State	???

PlayedFor						
<u>id</u>	<u>tid</u>	<u>year</u>	position	jersey	games	starts
20000	1024	1991	LB	52	16	0
...						
20000	1024	1995	LB	52	16	16
20000	1024	1996	LB	52	15	15
20000	1025	1997	LB	52	16	16
...						
20000	1030	2001	LB	51	15	0
20000	1030	2002	LB	95	6	1
20000	1024	2002	LB	52	3	0
20001	1007	1991	DT	96	11	3



3rd Normal Form (3NF)

A table is in 3NF if it is in 1NF, 2NF, and *no non-primary-key attribute is transitively dependent on the primary key*. Transitive dependence occurs when through a series of FDs a primary key implies a correlation between non-key elements. This can happen in one of two ways.



Various forms of transitive dependence in a list of functional dependences



3rd Normal Form

These FDs don't actually have nice transitive dependencies, and, this table is still full of redundancies. But, we can modify a FD, and if we are willing to accept it we can significantly reduce the redundancy, and simplify many common joins.

~~$TY \rightarrow LM, MY \rightarrow L$~~

$T \rightarrow M, MY \rightarrow L (TY \rightarrow M, TY \rightarrow L)$

Team			
<u>tid</u>	<u>year</u>	location	mascot
1024	1991	Los Angeles	Raiders
...			
1024	1995	Oakland	Raiders
1024	1996	Oakland	Raiders
1025	1997	St Louis	Rams
...			
1030	2001	Pittsburgh	Steelers
1030	2002	Pittsburgh	Steelers
1024	2002	Oakland	Raiders
1007	1991	Pheonix	Cardinals



Team	
<u>tid</u>	mascot
1024	Raiders
...	
1025	Rams
...	
1030	Steelers
...	
1007	Cardinals

TeamLocation		
<u>tid</u>	<u>year</u>	location
1024	1991	Los Angeles
...		
1024	1995	Oakland
1024	1996	Oakland
1025	1997	St Louis
...		
1030	2001	Pittsburgh
1030	2002	Pittsburgh
1024	2002	Oakland
1007	1991	Pheonix



Boyce-Codd Normal Form

A relation is in BCNF, iff every set of determinant attributes (left-hand-side of an FD) is a candidate key of some relation.

$I \rightarrow \text{PHWCD}$, $\text{PCD} \rightarrow I$,

$\underline{\text{ITY}} \rightarrow \text{PJGS}$, $\text{JTY} \rightarrow I$,

$\underline{T} \rightarrow M$, $\text{MY} \rightarrow L$,

$\underline{\text{TY}} \rightarrow L$

In our case
this is true,
thus, we are
already in
BCNF.

Team	
<u>tid</u>	mascot
1024	Raiders
1025	Rams
1030	Steelers
1007	Cardinals

TeamLocation		
<u>tid</u>	<u>year</u>	location
1024	1991	Los Angeles
...		
1024	1995	Oakland
1024	1996	Oakland
1025	1997	St Louis
...		
1030	2001	Pittsburgh
1030	2002	Pittsburgh
1024	2002	Oakland
1007	1991	Pheonix

Player					
<u>id</u>	name	height	weight	college	dob
20000	Mike Jones	6-1	240	Missouri	1969-04-15
20001	Mike Jones	5-11	181	NC State	???

PlayedFor						
<u>id</u>	<u>tid</u>	<u>year</u>	position	jersey	games	starts
20000	1024	1991	LB	52	16	0
...						
20000	1024	1995	LB	52	16	16
20000	1024	1996	LB	52	15	15
20000	1025	1997	LB	52	16	16
...						
20000	1030	2001	LB	51	15	0
20000	1030	2002	LB	95	6	1
20000	1024	2002	LB	52	3	0
20001	1007	1991	DT	96	11	3



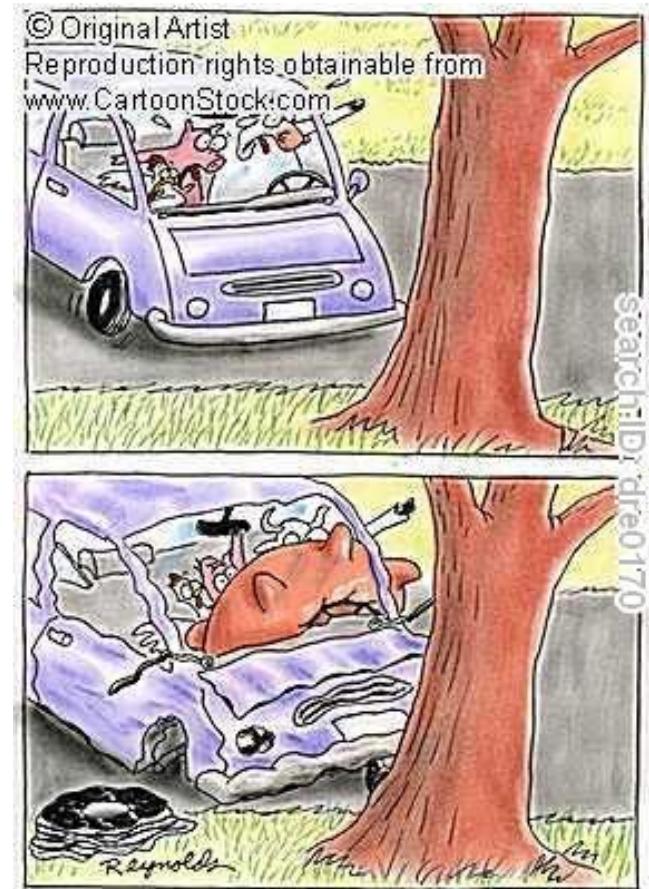
Summary of Schema Refinement

- ❖ If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- ❖ If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
 - Must consider whether all FDs are preserved. If a lossless-join, dependency preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), should consider decomposition into 3NF.
 - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.



Database Crash Recovery

PS #3 graded
Move PS #4 due on 11/7





Review: The ACID properties

- ❖ **Atomicity:** All actions of a transaction happen, or none happen.
- ❖ **Consistency:** If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- ❖ **Isolation:** Execution of one Xact is isolated from that of other Xacts.
- ❖ **Durability:** If a Xact commits, its effects persist.
- ❖ The **Recovery Manager** guarantees Atomicity & Durability.

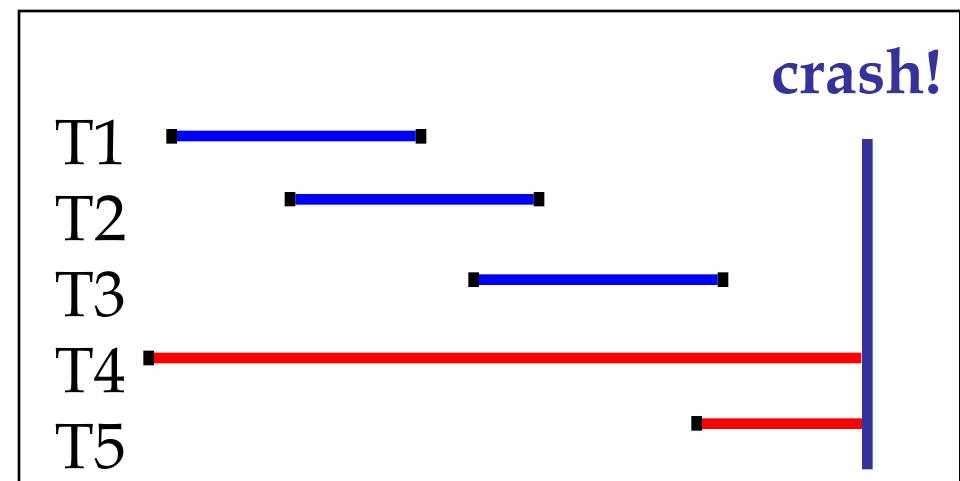


Motivation

- ❖ Atomicity:
 - Transactions may abort (“Rollback”).
- ❖ Durability:
 - What if DBMS Crashes?
 (“Worse case”, a few unfinished Xacts are lost)

Desired state after system restarts?

- T1, T2 & T3 should be **durable**.
- T4 & T5 should be **aborted** (no effect).





Assumptions

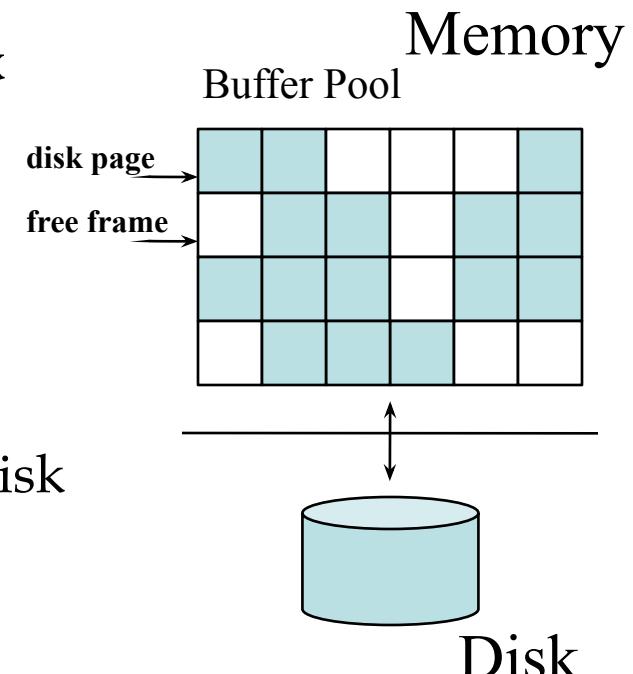
- ❖ Concurrency control is in effect.
 - In particular, locks are acquired on blocks before reading or writing and are released after commit.
- ❖ Updates are happening “in place”.
 - i.e. data is overwritten on (or deleted from) non-volatile disk.
 - “In place” implies, we are not using a temporary/in memory database or cache, but one that is persistent.
- ❖ Can you think of a simple scheme to guarantee Atomicity & Durability?



Recalling the Buffer Pool

Which of the following types of pages might be found in the buffer pool?

- A) Interior steering nodes of a B⁺-tree index
- B) Intermediate sorted pages from a recent sort-merge-join
- C) A bucket of <key, rid> pairs from a hash index
- D) A “dirty” updated page from a relation that has yet to be committed to disk
- E) All of the above



Of these, which must be tracked in by the log?



Handling the Buffer Pool

- ❖ **Force** every write to disk? Stall DBMS until completed

- Poor response time.
- But provides durability.

- ❖ **Steal** buffer-pool frames from uncommitted Xacts?
(flush dirty frames, only when a new frame is needed)

- If not, poor throughput (multiple writes to same page).
- If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

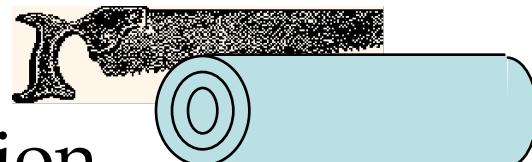


More on Steal and Force

- ❖ **STEAL** (why enforcing Atomicity is hard)
 - What if a page, P, dirtied by some unfinished Xact is written to disk to free up a buffer slot, and the Xact later aborts?
 - Must remember the old value of P at steal time (to **UNDO** the page write).
- ❖ **NO FORCE** (why enforcing Durability is hard)
 - What if system crashes before a page dirtied by a committed Xact is flushed to disk?
 - Write as little as possible, in a convenient place, at commit time, to support **REDO**ing modifications.



Basic Idea: Logging



- ❖ Record sufficient information to REDO and UNDO every change in a *log*.
 - Write and Commit sequences saved to log (on a separate disk or replicated on multiple disks).
 - Minimal info (diff) written to log, so multiple updates fit in a single log page.
- ❖ Log: An ordered list of REDO/UNDO actions
 - Log record contains:
 $\langle \text{XID}, \text{pageID}, \text{offset}, \text{length}, \text{old data}, \text{new data} \rangle$
 - and additional control info (which we'll see soon).



Write-Ahead Logging (WAL)

Key Idea of WAL: *Before writing any page to disk, every update log that describes any previous change to this page must be forced to stable storage.*

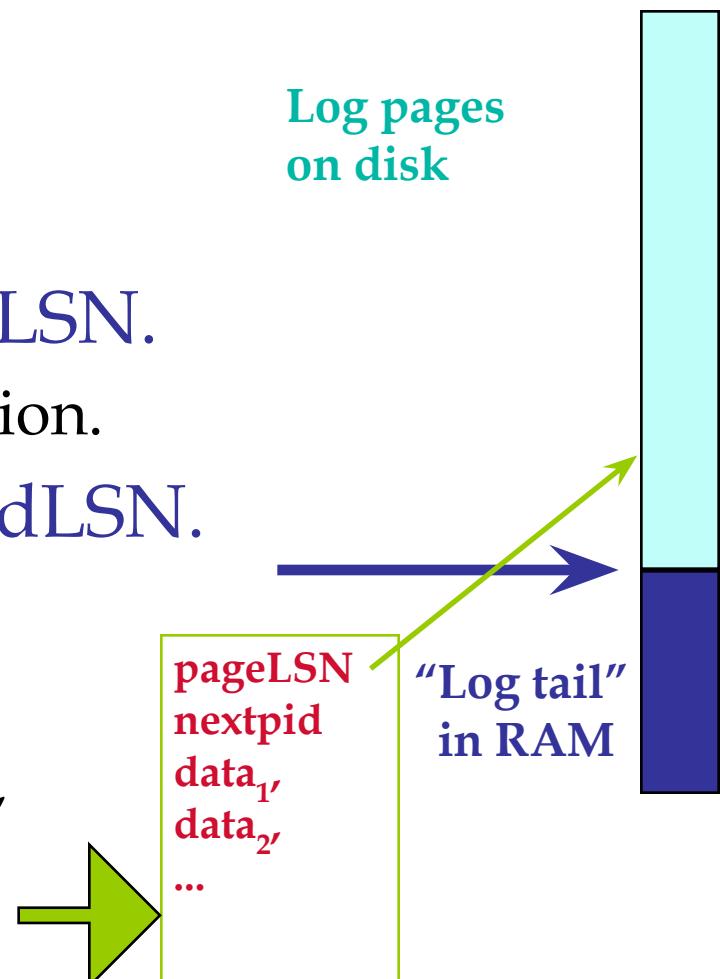
- ❖ The Write-Ahead Logging Protocol:
 1. Modifications of database objects must *first* be recorded in the log, and the log updated, *before* any change to the actual object
 2. Must write all log records of a Xact *before it commits.*
- ❖ #1 guarantees Atomicity.
- ❖ #2 guarantees Durability.
- ❖ Exactly how is logging (and recovery!) done?
 - We'll study the ARIES algorithm.



WAL & the Log



- ❖ Each log record has a unique Log Sequence Number (LSN).
 - LSNs are always increasing.
- ❖ Each *data page* contains a pageLSN.
 - LSN of its most recent modification.
- ❖ System keeps track of a flushedLSN.
 - Max LSN flushed from the page buffer so far.
- ❖ WAL: *Before* a page is written,
 - $\text{pageLSN} \leq \text{flushedLSN}$





Log Records

LogRecord fields:

prevLSN
XID
type
pageID
length
offset
before-image
after-image

update
records
only

Possible log record types:

- ◆ **Update**
- ◆ **Commit**
- ◆ **Abort**
- ◆ **End** (signifies end of commit or abort)
- ◆ **Compensation Log Records (CLRs)**
 - for UNDO actions

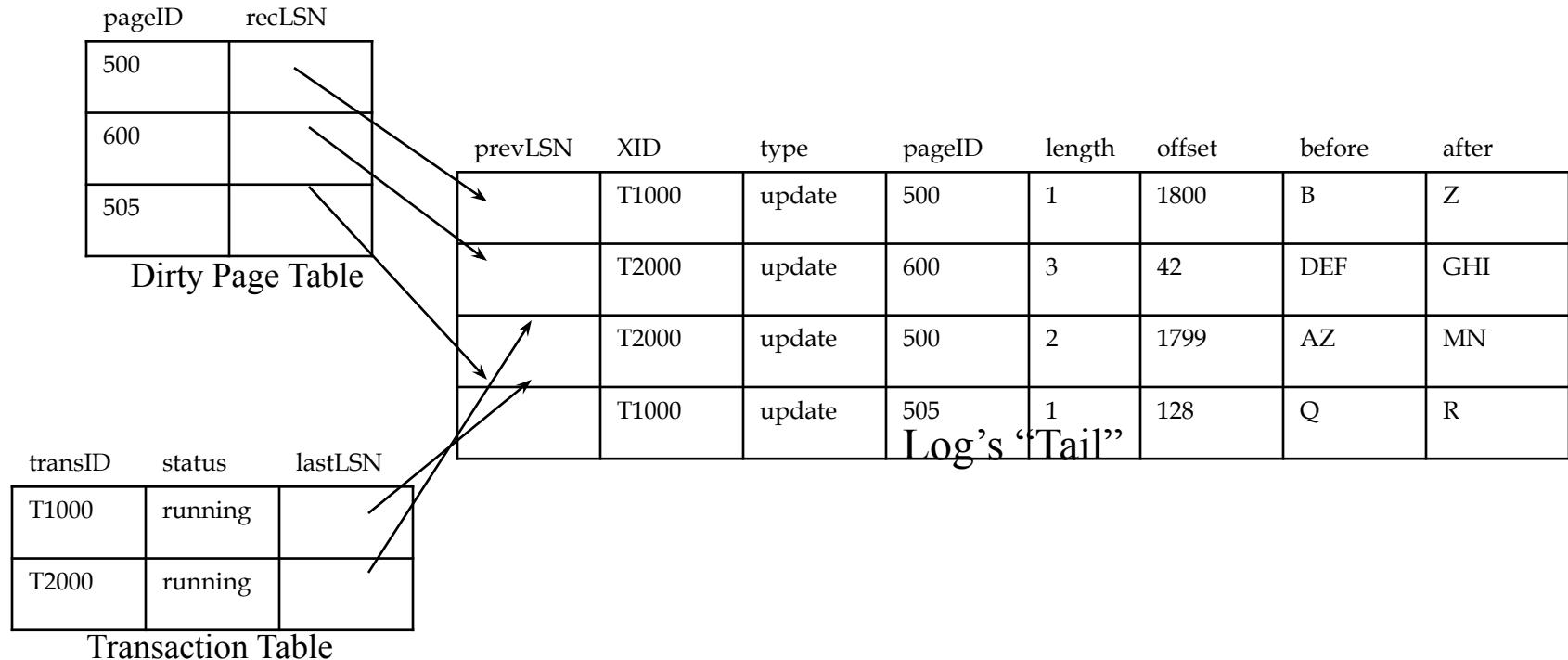


Other Log-Related State

- ❖ **Transaction Table:**
 - One entry per active Xact.
 - Contains **XID**, **status** (running/committed/aborted), and **lastLSN** due to Xact
- ❖ **Dirty Page Table:**
 - One entry per dirty page in buffer pool
 - Contains **recLSN** -- the LSN of the log record which ***first*** dirtied the page



Log and Table Entries





Normal Execution of an Xact

- ❖ Series of **reads & writes**, terminated by **commit** or **abort**.
 - We will assume that write is atomic on disk.
 - In practice, additional details to deal with non-atomic writes.
- ❖ Strict 2PL.
- ❖ STEAL, NO-FORCE buffer management, with Write-Ahead Logging.



Checkpointing

- ❖ Periodically, the DBMS creates a checkpoint, to minimize recovery time in the event of a system crash. What is written to log and disk:
 - begin_checkpoint record: Indicates when chkpt began.
 - end_checkpoint record: Contains current active *Xact table* and *dirty page table*. This is a “fuzzy checkpoint”:
 - Xacts continue to run; so these tables are accurate only as of the time of the begin_checkpoint record.
 - No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page.
(So it's a good idea to periodically flush dirty pages to disk!)
 - Store LSN of chkpt record in a safe place (master record).

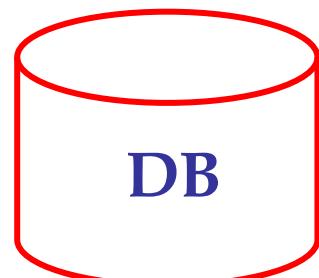


The Big Picture: What's Stored Where



LogRecords

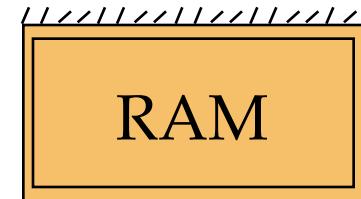
prevLSN
XID
type
pageID
length
offset
before-image
after-image



Data pages

each
with a
pageLSN

master record



Xact Table

lastLSN
status

Dirty Page Table

recLSN

flushedLSN

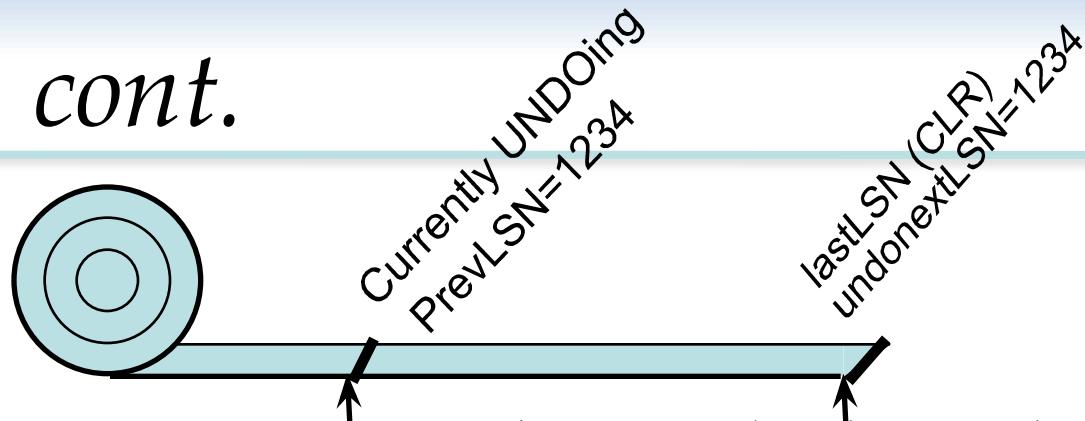


Simple Transaction Abort

- ❖ For now, consider an explicit abort of a Xact.
 - No crash involved.
- ❖ We want to “play back” the log in reverse order, UNDOing updates.
 - Get *lastLSN* of Xact from Xact table.
 - Can follow chain of log records backward via the *prevLSN* field.
 - Before starting UNDO, write an *Abort* log record.
 - For recovering from crash during UNDO!



Abort, cont.



- ❖ To perform UNDO, must have a lock on data!
- ❖ Before restoring old value of a page, write a Compensation Log Record (CLR):
 - Continue logging while you UNDO!!
 - CLR has one extra field: **undonextLSN**
 - Points to the next LSN to undo (prevLSN of log entry)
 - CLRs are *never* Undone (but they might be Redone when repeating history: guarantees Atomicity!)
- ❖ At end of UNDO, write an “end” log record.

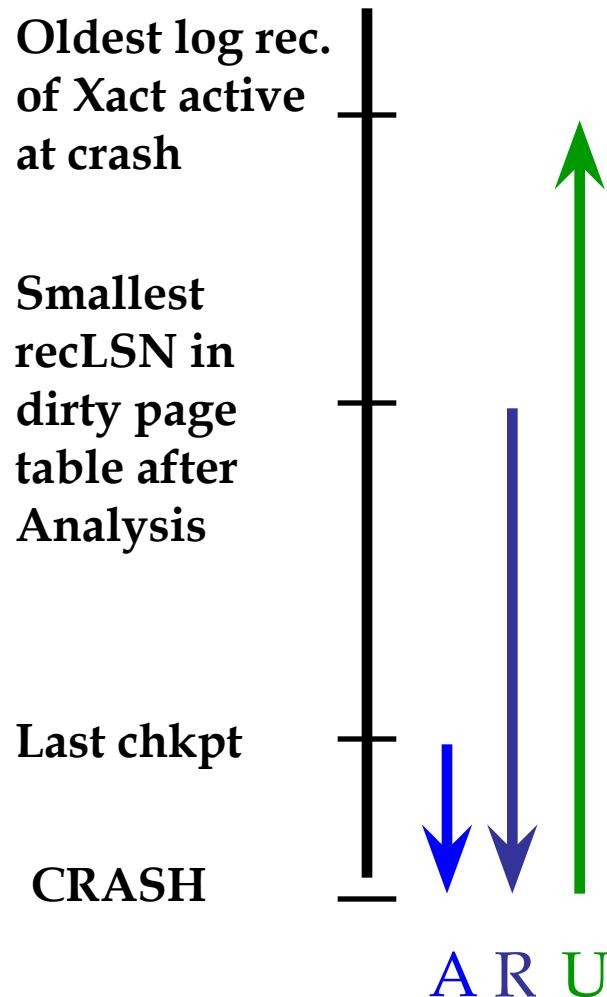


Transaction Commit

- ❖ Write **commit** record to log.
- ❖ All log records up to Xact's **lastLSN** are flushed on a commit.
 - Guarantees that **flushedLSN \geq lastLSN**.
 - Note that log flushes are sequential, synchronous writes to disk.
 - Many log records per log page.
- ❖ Commit() returns.
- ❖ Write **end** record to log.



Crash Recovery: Big Picture



- Start from a **checkpoint** (found via **master** record).
- ARIES 3 phases. Need to:
 - **Analysis:** Figure out which Xacts committed since last checkpoint, and which did not finish.
 - **REDO** *all* logged actions.
Repeats “writing” history to recreate buffer pool
 - **UNDO** effects of unfinished “loser” Xacts.



Recovery: The Analysis Phase

- ❖ Reconstruct state at checkpoint.
 - via the `end_checkpoint` record.
- ❖ Scan log forward from checkpoint.
 - Look for `End` records: Remove Xact from Xact table because it safely completed.
 - Other records: Add Xact to Xact table, set `lastLSN=LSN`, change Xact status on `commit`.
 - Update record: If P not in Dirty Page Table,
 - Add P to D.P.T., set its `recLSN=LSN`.



Recovery: The REDO Phase

- ❖ We *repeat History* to reconstruct state at crash:
 - Reapply *all* updates (even of aborted Xacts!), redo CLRs.
- ❖ Scan forward from log record of the smallest **recLSN** in the dirty page table. For each CLR or update log rec **LSN**, REDO the action unless:
 - Affected page is not in the Dirty Page Table, or
 - Affected page is in D.P.T., but has **recLSN > LSN**, or
 - **pageLSN** (in DB) \geq **LSN**.
- ❖ To **REDO** an action:
 - Reapply logged changes (restore to before state).
 - Set **pageLSN** to **LSN**. No additional logging!



Recovery: The UNDO Phase

$\text{ToUndo} = \{ l \mid l \text{ a lastLSN of a "loser" Xact}\}$

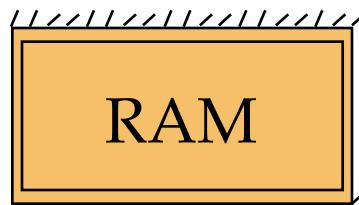
Repeat:

- Choose largest LSN among ToUndo.
- If this LSN is a CLR and `undonextLSN==NULL`
 - Write an End record for this Xact.
- If this LSN is a CLR, and `undonextLSN != NULL`
 - Add `undonextLSN` to ToUndo
- Else this LSN is an update. UNDO the update, write a CLR, add `prevLSN` to ToUndo.

Until ToUndo is empty.



Example of Recovery



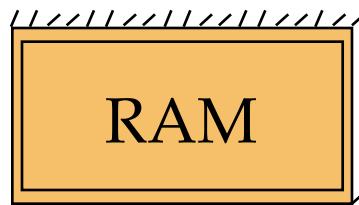
Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo

LSN	LOG
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update T2 writes P3
30	T1 abort
40	CLR: Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5
	CRASH, RESTART



Example: Crash During Restart!



Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo

LSN	LOG
00,05	+ begin_checkpoint, end_checkpoint
10	+ update: T1 writes P5
20	+ update T2 writes P3
30	- T1 abort
40,45	- CLR: Undo T1 LSN 10, T1 End
50	+ update: T3 writes P1
60	+ update: T2 writes P5
65	X CRASH, RESTART
70	- CLR: Undo T2 LSN 60
80,85	- CLR: Undo T3 LSN 50, T3 end
85	X CRASH, RESTART
90	- CLR: Undo T2 LSN 20, T2 end



Additional Crash Issues

- ❖ What happens if system crashes during Analysis? During REDO?
- ❖ How to limit the amount of work in REDO?
 - Flush dirty pages asynchronously in the background.
 - Watch out for “hot spots”!
- ❖ How to limit the amount of work in UNDO?
 - Avoid long-running Xacts.



Summary of Logging/Recovery

- ❖ Recovery Manager guarantees Atomicity & Durability.
- ❖ Uses WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- ❖ LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- ❖ pageLSN allows comparison of data page and log records.



Summary, Cont.

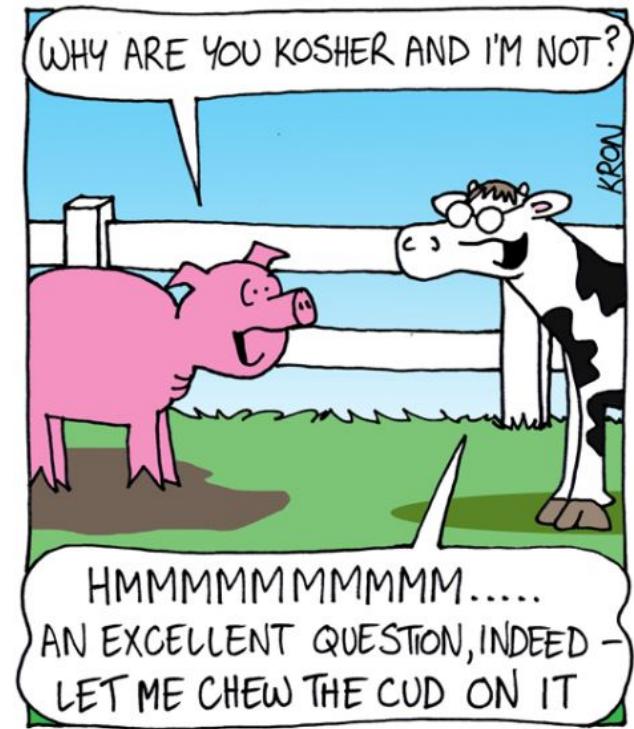
- ❖ **Checkpointing:** A quick way to limit the amount of log to scan on recovery.
- ❖ Recovery works in 3 phases:
 - **Analysis:** Forward from checkpoint.
 - **Redo:** Forward from oldest recLSN.
 - **Undo:** Backward from end to first LSN of oldest Xact alive at crash.
- ❖ Upon Undo, write CLRs.
- ❖ Redo “repeats history”: Simplifies the logic!



Intro to NoSQL Databases

PS #4 due before midnight tonight

PS #5 will be issued tonight



The CARTOON KRONICLES



Structured vs Unstructured

Data can be broadly classified into types:

1. Structured data:

- a. Conforms to a predefined model, which organizes data into a form that is relatively uniform and, thus, easy to store, process, retrieve and manage.
- b. e.g. rows with common attributes (relational data)

2. Unstructured data:

- a. Opposite of structured data
- b. e.g. flat binary files containing text, video, or audio

Note: data is not completely devoid of a structure (e.g., an audio file may still have an encoding structure and some metadata associated with it, text often has abstracts, intros, and references).



Dynamic vs. Static

Data can also be classified by temporal significance

Dynamic Data:

Data that changes relatively frequently

e.g., How many steps Joe has walked today, live statistics of a sporting event, or financial object

Static Data:

Opposite of dynamic data

e.g., Medical imaging data from MRI or CT scans

Historical sports records



Data Classifications

Segmenting data according to one of the following 4 quadrants can help in designin, developing, and maintaining effective storage solutions

	Dynamic	Static
Structured	Bank Transactions, Financial Statistics	Historical Sports Statistics, College Transcripts
Unstructured	Live video streams, On-line shared documents, Message Feeds	Archived YouTube videos, Warehoused medical data

Relational databases were designed for structured data

File systems or *NoSQL databases* can be used for (static), unstructured data (*more on these later*)



Scaling Issues of Data Access

Traditional DBMSs can be either scaled:

- ❖ **Vertically (or Up)**

- Achieved by hardware upgrade
(e.g., faster CPUs, more memory, or larger disks)
- Limited by the amount of CPU, RAM and disk and network bandwidth available to a single machine

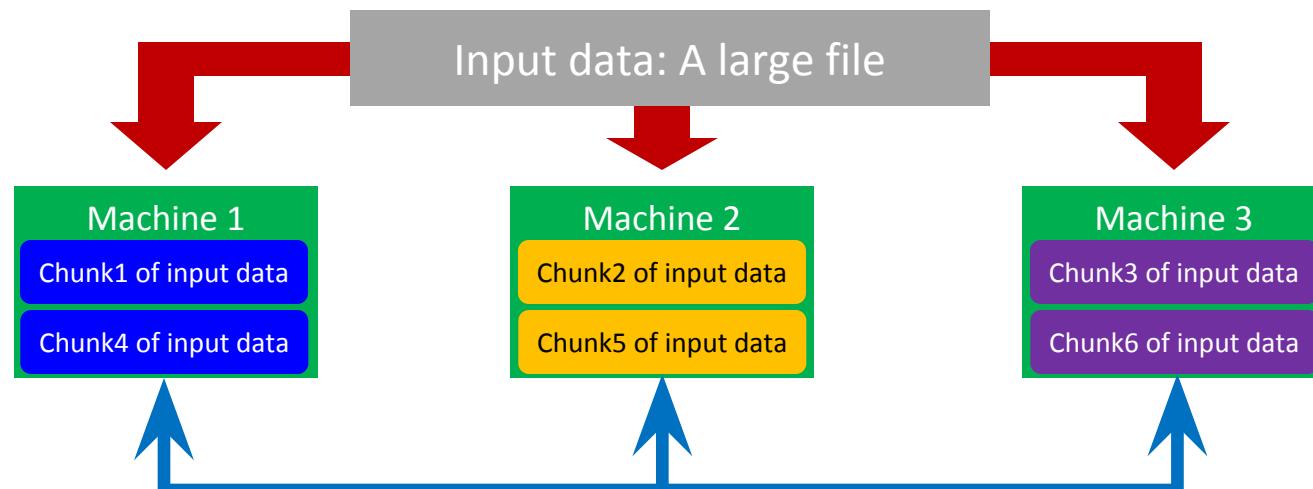
- ❖ **Horizontally (or Out)**

- Can be achieved by adding more machines
- Requires distributing databases and probably replication
 - Distribute tables to different machines
 - Distribute rows of tables to different machines
 - Distribute columns of tables to different machines
- Limited by the Read-to-Write ratio and communication overhead



Distributing Rows

Performance can be achieved by distributing the rows of tables across multiple DBMS servers. This is often called *sharding*. Sharding provides concurrent/parallel access.



E.g., Chunks 1, 5, and 3 can be queried in parallel



Computational Limits

Recall Amdahl's Law...

Suppose that the sequential execution of a program takes T_1 time units and the parallel execution on p processors/machines takes T_p time units

Suppose that out of the entire execution of the program, some fraction, s , is not parallelizable (s is for serial) while $1-s$ fraction is parallelizable.

Then the speedup by Amdahl's formula:

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$



An Example

- Suppose that:
 - 60% of your query can be parallelized
 - 6 machines are used in the parallel components of the tuple selection
- The speedup you can get according to Amdahl's law is:

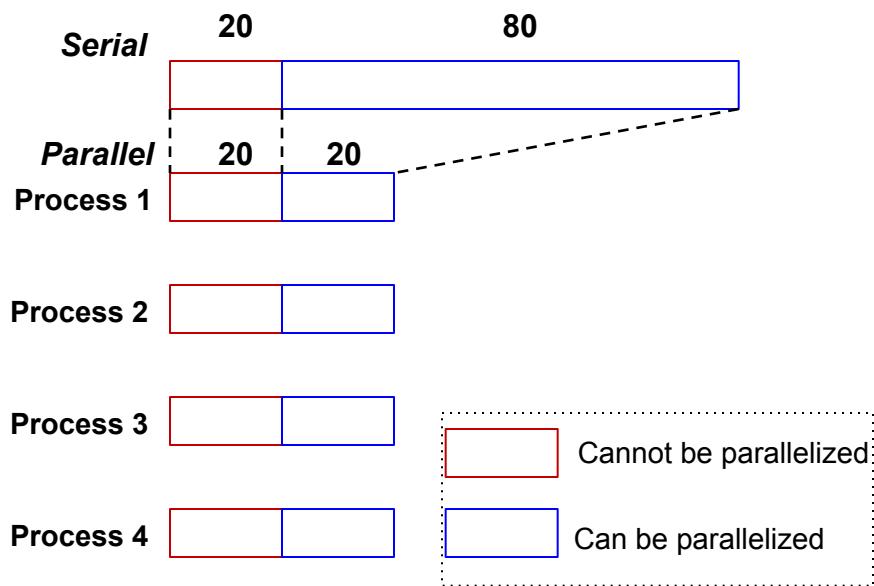
$$\frac{1}{s + \frac{1-s}{p}} = \frac{1}{0.4 + \frac{0.6}{6}} = 2.0$$

Although you use 6 processors you do not get a speedup more than 2 times!

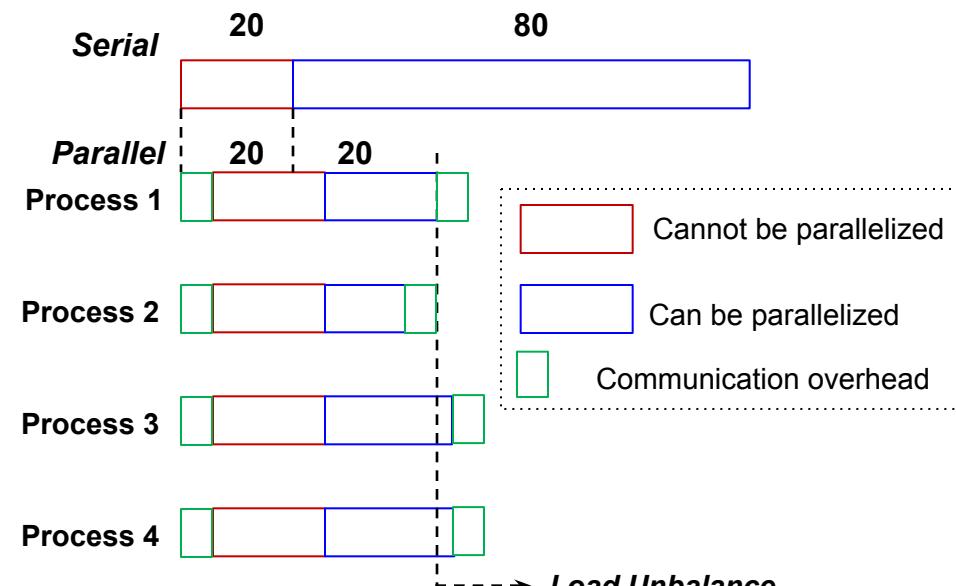


Communication & Imbalance

- In reality, Amdahl's argument is over simplified
- Communication overhead and potential workload imbalance also impact parallel programs



1. Parallel Speed-up: The "Ideal" Case (2.5x)



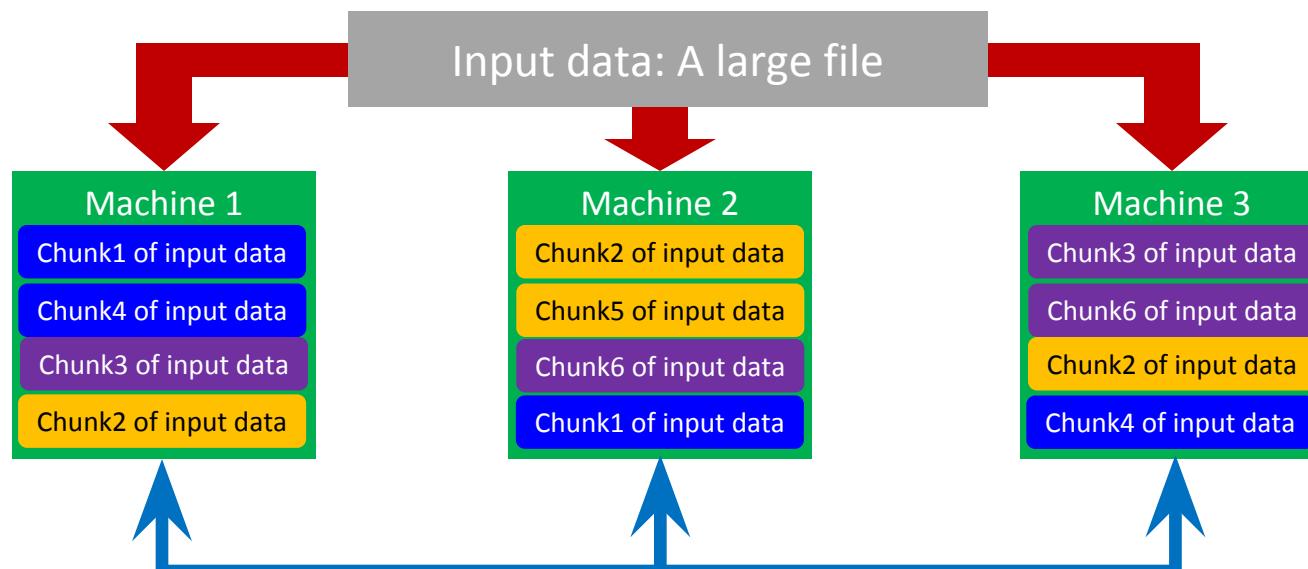
2. Parallel Speed-up: Actual Case (~2X)



Shard and Replicate

Why replicate data?

- Replicating data across servers helps by:
 - Avoiding performance bottlenecks
 - Avoiding single point of failures

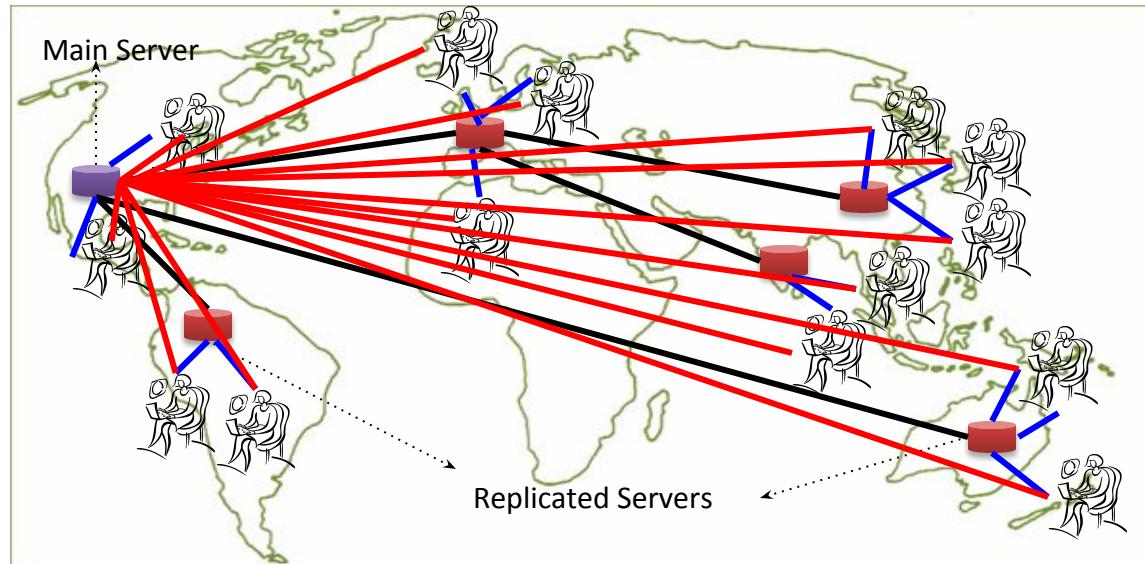




Shard and Replicate

Why replicate data?

- Replicating data across servers helps by:
 - Avoiding performance bottlenecks
 - Avoiding single point of failures
 - Also enhances *scalability* and *availability*

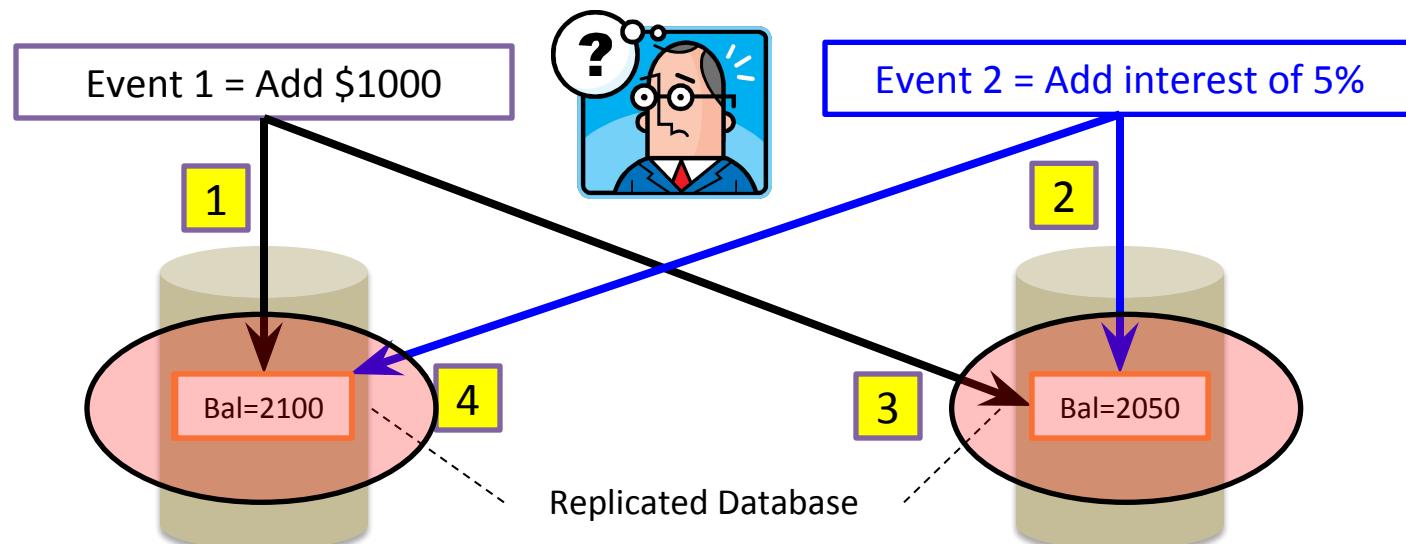




But,

Consistency Becomes a Challenge...

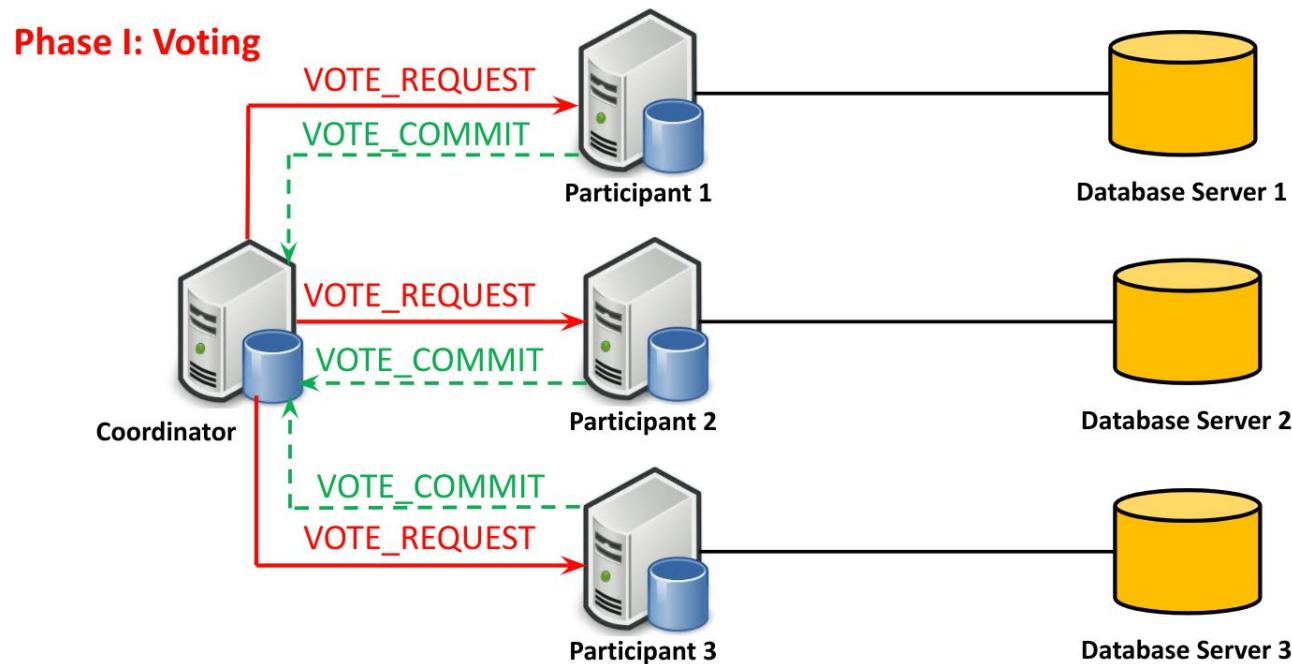
- An example:
 - In an e-commerce application, the bank database has been replicated across two servers
 - Maintaining consistency of replicated data is a challenge
 - Our scheduling approach actually assumes a serial execution...





Distributed 2PL

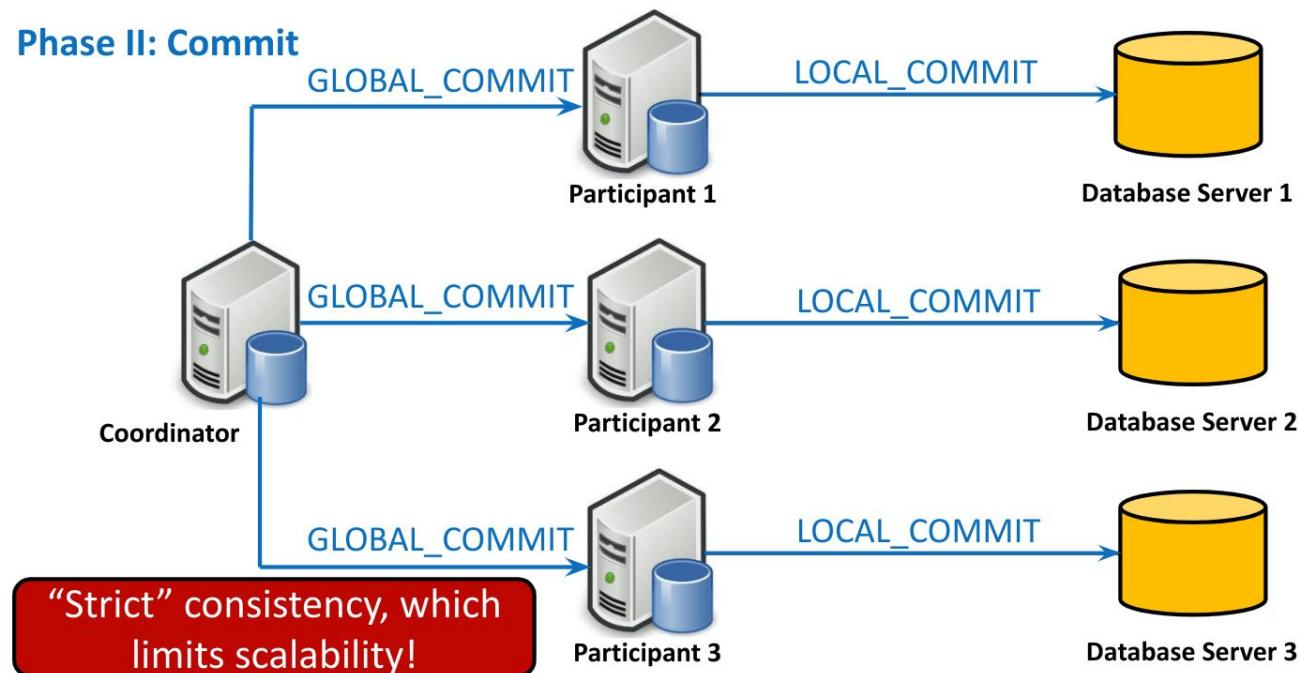
- Two-phase locking protocol (2PL) can still be used to ensure atomicity and consistency, but it increases the serial fraction of execution, and usually involves a single "lock-authority" or coordinator.





Distributed 2PL

- Two-phase locking protocol (2PL) can still be used to ensure atomicity and consistency, but it increases the serial fraction of execution, and usually involves a single "lock-authority", coordinator, and voting.





The CAP Theorem

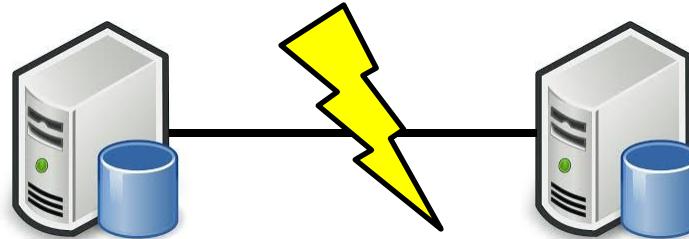
- The limitations of distributed databases can be described in the so called the **CAP theorem**
 - **Consistency**: every node always sees the same data at any given instance (i.e., strict consistency)
 - **Availability**: continues to operate, even if nodes in a cluster crash, or some hardware or software parts are down due to upgrades
 - **Partition Tolerance**: continues to operate in the presence of network partitions (breaks in connectivity)

CAP theorem: any distributed database with shared data, can have at most two of the three desirable properties, C, A or P



CAP Examples

- Assume two nodes on opposite sides of a network partition:



- Availability + Partition Tolerance forfeit Consistency
- Consistency + Partition Tolerance entails that one side of the partition must act as if it is unavailable, thus forfeiting Availability
- Consistency + Availability is only possible if there is no network partition, thereby forfeiting Partition Tolerance



Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
 - A few minutes of downtime means significant lost revenue
- When scaling databases to 1000s of machines, the likelihood of a node or a network failure increases tremendously
- Therefore, in order to have strong guarantees on *Availability* and *Partition Tolerance*, they had to sacrifice “strict” Consistency (*as implied by the CAP theorem*)



The Consistency Trade-off

- Maintain a balance between the strictness of consistency versus availability/scalability
- "Good-enough" consistency is application dependent



Performance is measured in throughput (how many transactions per second the system can manage) and latency (how long you have to wait)



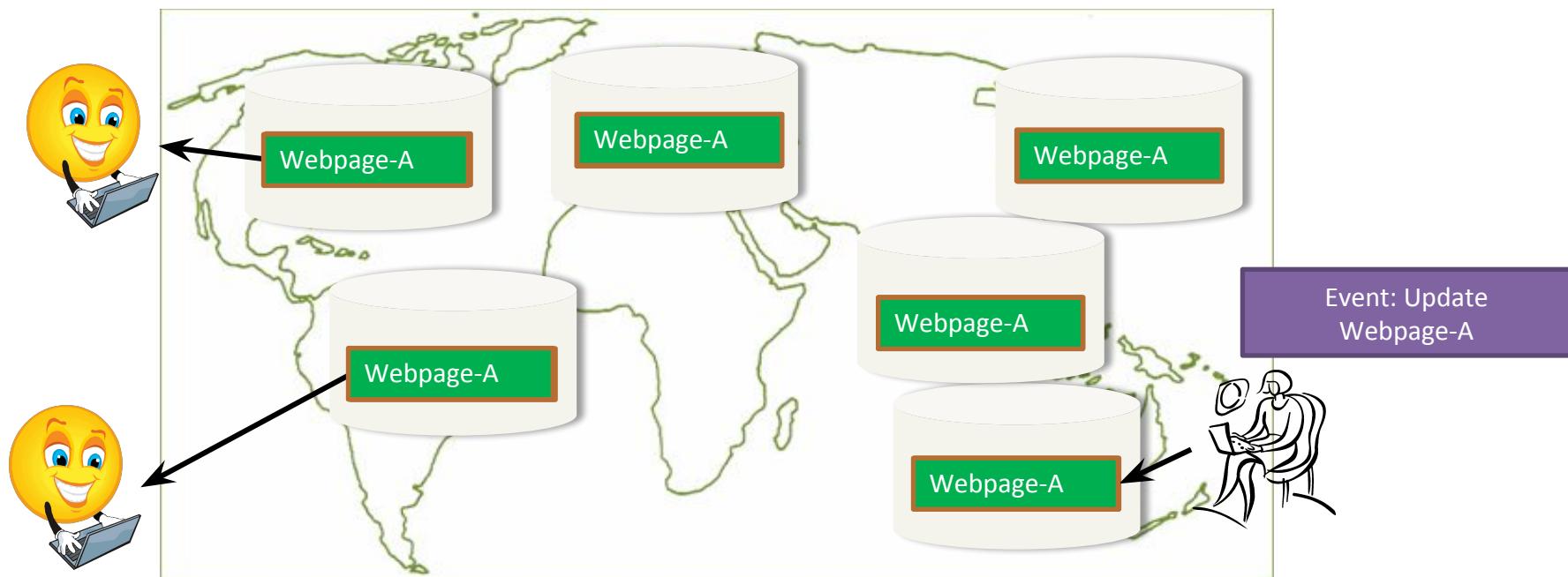
BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions
- This resulted in databases with relaxed **ACID** guarantees
- In particular, such databases apply the **BASE** properties:
 - **B**asically **A**vailable: the system guarantees availability
 - **S**oft-**S**tate: state of the system may change over time but might be slightly inconsistent for small intervals
 - **E**ventual **C**onsistency: the system will *eventually* become consistent



Eventual Consistency

- A database is termed as *Eventually Consistent* if:
 - All replicas will *gradually* converge to a single consistent state in the absence of updates for some specified interval





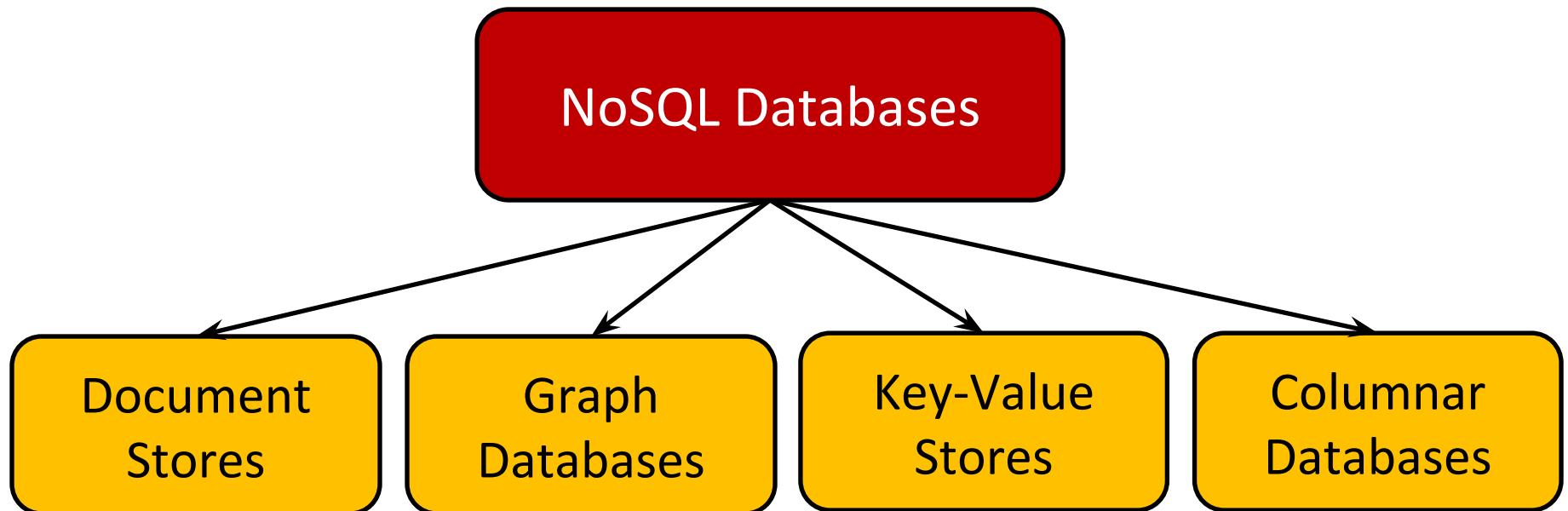
NoSQL Databases

- To this end, a new class of databases emerged, which mainly follow the BASE properties
 - These were dubbed as NoSQL databases
 - E.g., Amazon's Dynamo and Google's Bigtable
- Main characteristics of NoSQL databases include:
 - No strict schema requirements
 - No strict adherence to ACID properties
 - Availability > Consistency
 - Consistency eventually, if all updates stop



Types of NoSQL Databases

Here is a taxonomy of NoSQL databases:





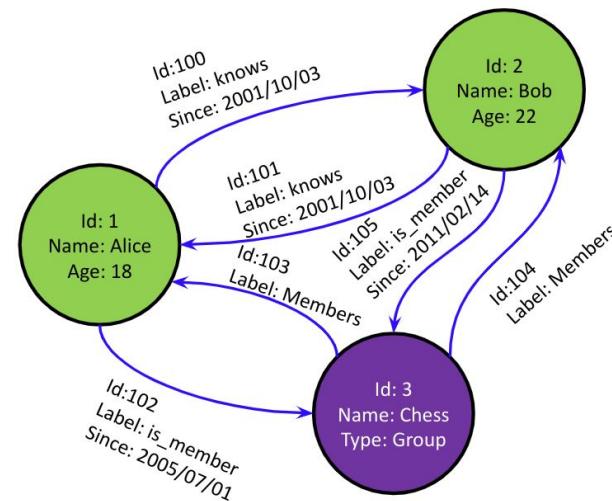
Document Stores

- Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
 - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
 - This allows document stores to outperform typical file systems
- e.g., MongoDB and CouchDB (both can be queried using MapReduce (more on this next time!))



Graph Databases

- Data are represented as vertices and edges
 - Vertices are like ER "entities"
 - Edges are like ER "relations"



- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)
- E.g., Neo4j and VertexDB



Key-Value Stores

- Keys are mapped to (possibly) more complex value (e.g., lists)
- Keys can be stored in a hash table and can be distributed easily
- Such stores typically support regular CRUD (create, read, update, and delete) operations
 - They don't support joins or aggregate functions
- E.g., Amazon DynamoDB and Apache Cassandra



Columnar Databases

- Columnar databases are a hybrid of DBMSs and Key-Value stores
 - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)

record ₁	Alice	42	NC
record ₂	Bob	35	CA
record ₃	Carol	25	CA

Row-Order

column ₁	Alice	Bob	Carol
column ₂	42	35	25
column ₃	NC	CA	CA

Columnar or
Column-Order

group1	Alice	Bob	Carol			
group2	42	NC	35	CA	25	CA

Columnar with
Groups

- Values are queried by matching keys, to find column indices
- E.g., HBase and Vertica

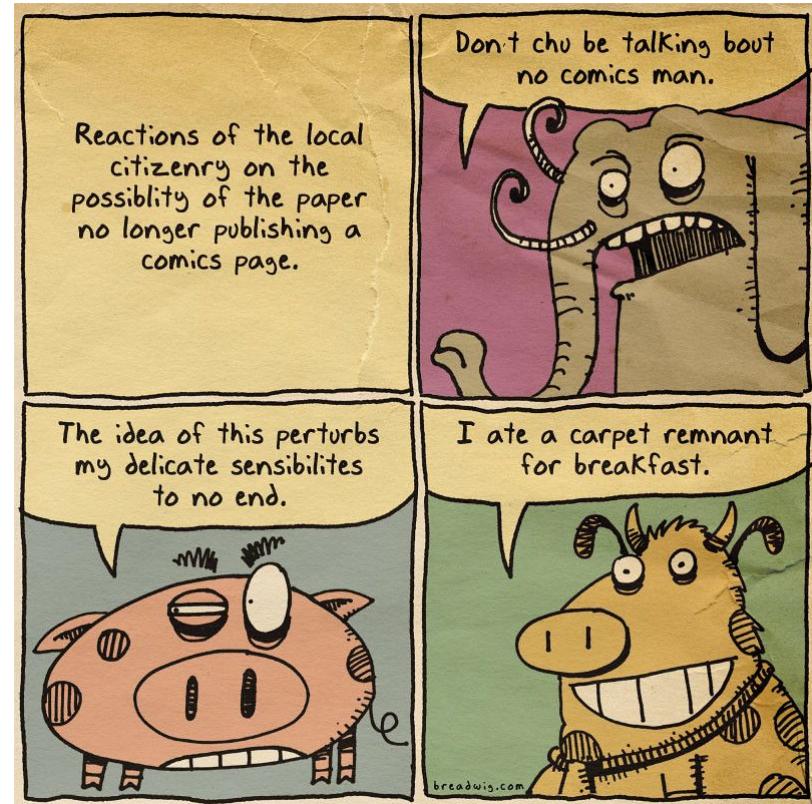


Summary

- Data can be classified into 4 types, *structured*, *unstructured*, *dynamic* and *static*
- Databases can be scaled *up* or *out*
- Strict consistency limits scalability
- The *CAP theorem* states that any distributed database with shared data can have at most two of the three desirable properties: **Consistency**, **Availability**, **Partition Tolerance**
- CAP theorem lead to various designs of databases with *relaxed* ACID guarantees
- NoSQL (databases follow the BASE properties:
Basically Available, Soft-State, Eventual Consistency)
- NoSQL databases have different types:
Document Stores, Graph Databases, Key-Value Stores, Columnar Databases



MapReduce for Big Data





Distributed Big Data

- ❖ Google MapReduce
 - Motivation and History
 - Google File System (GFS)
 - MapReduce:
Schema, Example, MapReduce Framework
- ❖ Apache Hadoop
 - Hadoop Modules and Related Projects
 - Hadoop Distributed File System (HDFS)
 - Hadoop MapReduce
- ❖ Apache Spark



Big Data

- Big Data **analytics** (or data mining)
 - need to process **large** data **volumes** quickly
 - want to use computing **cluster** instead of a super-computer
 - Communication (**sending data**) between compute nodes is **expensive**
- ⇒ model of “move computing to data”



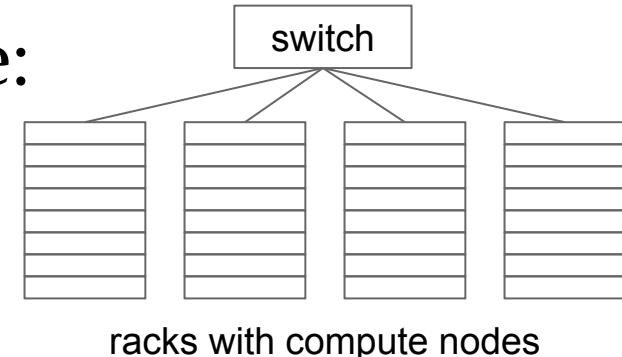
Big Data Processing

Computing **cluster** architecture:

1000s of computing nodes

10000s Gb of memory

10000s Tb of data storage



- HW **failures** are rather rule than exception, thus
 1. Files must be stored **redundantly**
 - over different **racks** to overcome also rack failures
 2. Computations must be divided into independent **tasks**
 - that can be **restarted** in case of a failure



MapReduce: Origins

- In 2003, Google had the following problem:
 1. How to **rank tens of billions** of webpages by their “importance” (PageRank) in a “reasonable” amount of time?
 2. How to **compute** these rankings **efficiently** when the data is scattered across **thousands of computers**?
- Additional factors:
 1. Individual data **files** can be enormous (**terabyte** or more)
 2. The files were **rarely updated**
 - the computations were **read-heavy**, but not very write-heavy
 - If **writes** occurred, they were **appended** at the end of the file



Google's Solution



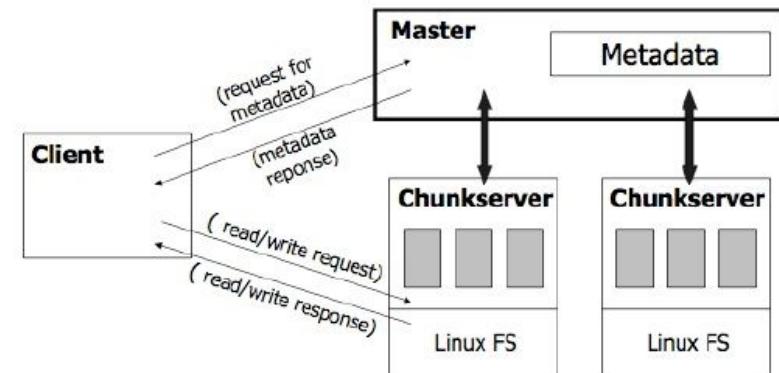
- Google found the following **solutions**:





Google File System (GFS)

- Files are divided into chunks (typically 64 MB)
 - The chunks are replicated at three different machines
 - The chunk size and replication factor are tunable
- One machine is a master, the other chunkservers
 - The master keeps track of all file metadata
 - mappings from files to chunks and locations of the chunks
 - To find a file chunk, client queries the master, and then contacts the relevant chunkservers
 - The master's metadata files are also replicated





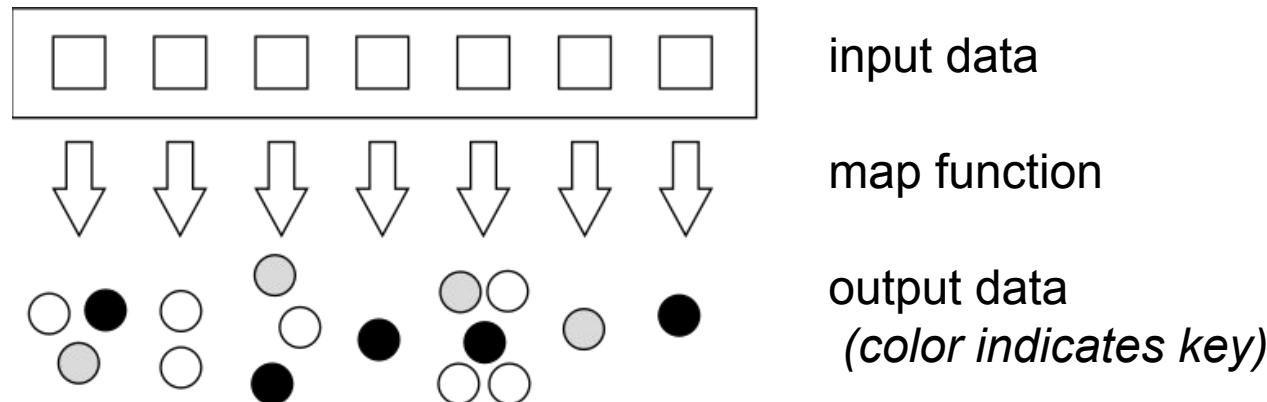
MapReduce

- ❖ MapReduce is a **programming model** sitting **on the top** of a Distributed File System
 - Originally: **no data model** – data is stored directly in **files**
- ❖ A **distributed** computational **task** has three phases:
 1. The **map** phase: data transformation
 2. The **grouping** phase
 - done automatically by the MapReduce Framework
 3. The **reduce** phase: data aggregation
- ❖ User must define **only** map & reduce **functions**



Map

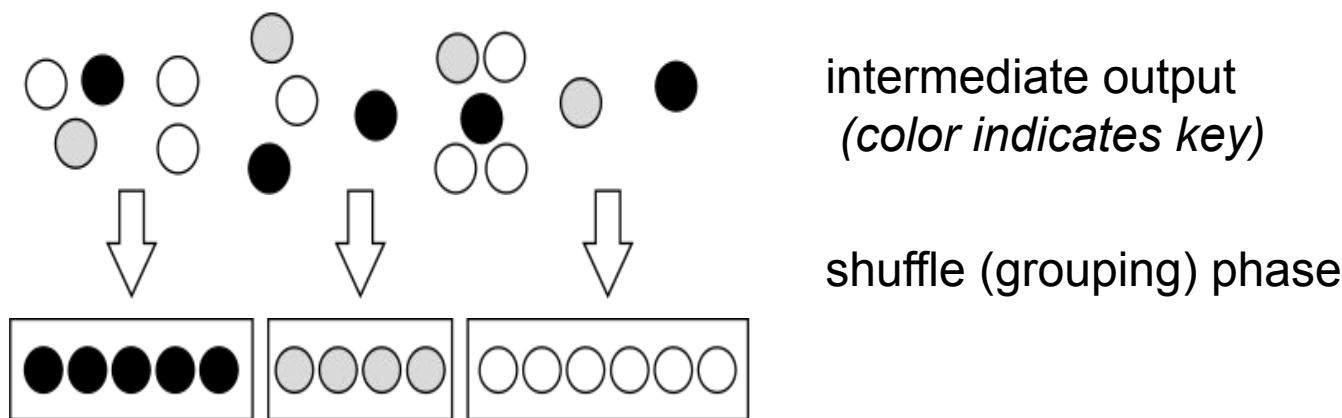
- ❖ **Map function** simplifies the problem in this way:
 - Input: a single data item (e.g. line of text) from a data file
 - Output: zero or more (key, value) pairs
- ❖ The **keys** are not typical “keys”:
 - They do **not** have to be **unique**
 - A map task can produce **several key-value pairs** with the same key (even from a single input)
- ❖ **Map phase** applies the map function to all items





Grouping Phase

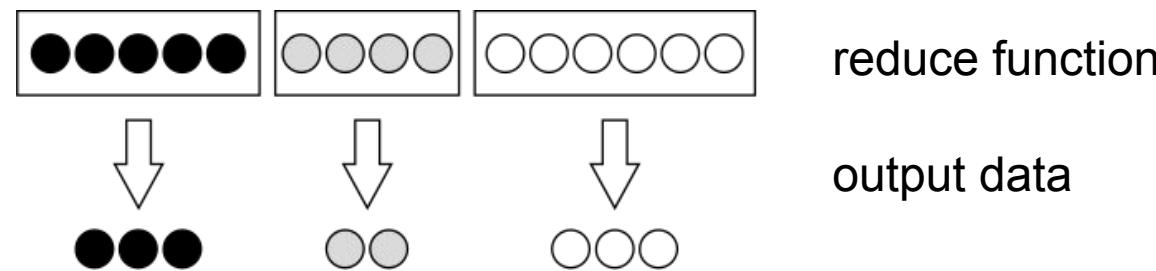
- ❖ **Grouping (Shuffling):** The key-value **outputs** from the **map** phase are grouped by key
 - Values sharing **the same key** are sent to the same reducer
 - These values are **consolidated** into a single list (key, list)
 - This is convenient for the reduce function
 - This phase is **realized by** the MapReduce **framework**





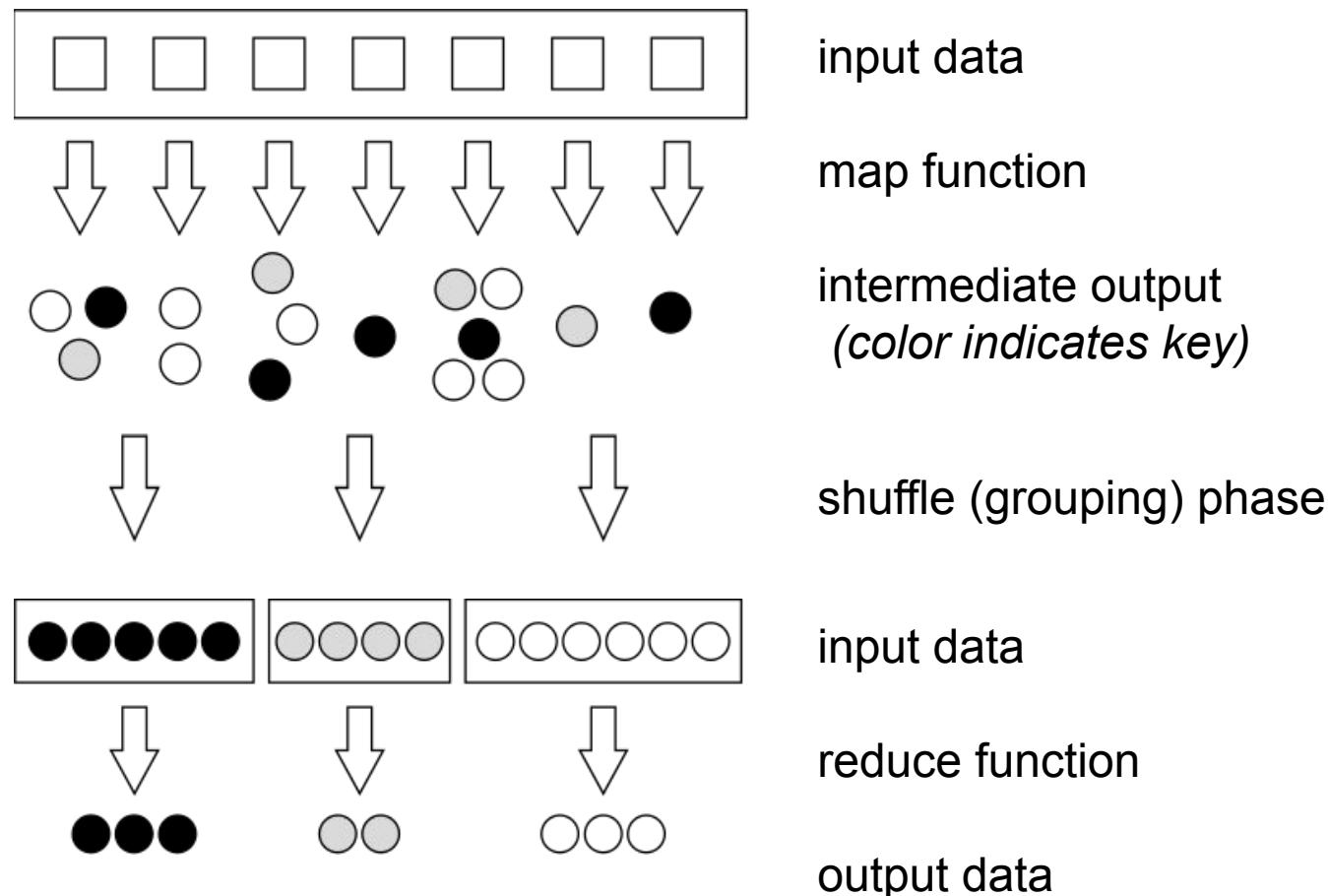
Reduce Phase

- ❖ Reduce: **combines** values with the same key
 - to achieve the **final result(s)** of the computational task
 - **Input:** (key, value-list)
 - value-list contains all values generated for given key in the Map phase
 - **Output:** (key, value-list)
 - zero or more **output records**





MapReduce, the full picture





Example: Word Count

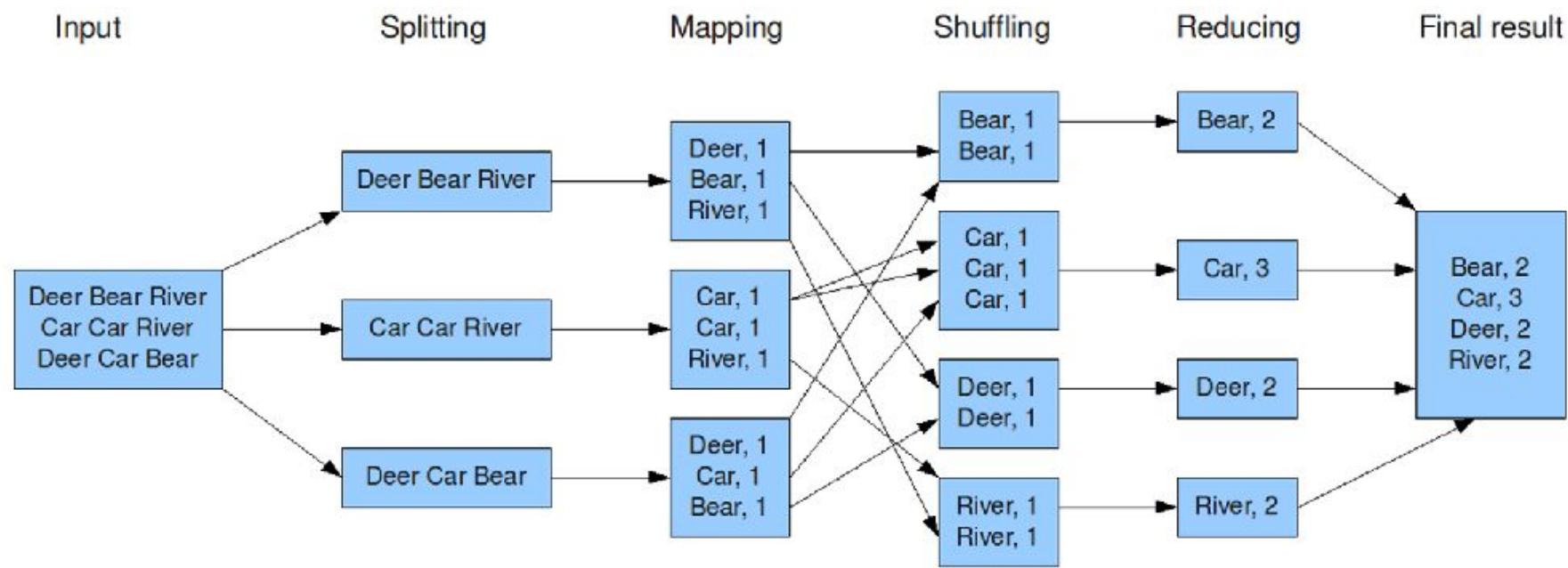
Task: Calculate **word frequency** in a set of documents

```
map(key, value):
    """ key: document name (ignored)
        value: content of document (words) """
    for w in value.split(' '):
        emitIntermediate(w, 1)

reduce(key, values):
    """ key: a word
        values: a list of counts """
    result = 0;
    for v in values:
        result += v
    emit(key, result)
```



Example: Word Count (2)





MapReduce: Combiner

- ❖ If the **reduce function** is **commutative** & **associative**
 - The values **can be combined** in any order and **combined per part (grouped)**
 - with the same result (e.g. Word Counts)
- ❖ ...then it opens space for **optimization**
 - Apply the **same reduce function** right after the map phase, **before shuffling** and redistribution to reducer nodes
- ❖ This (optional) step is known as the **combiner**
 - Note: it's still necessary to run the reduce phase



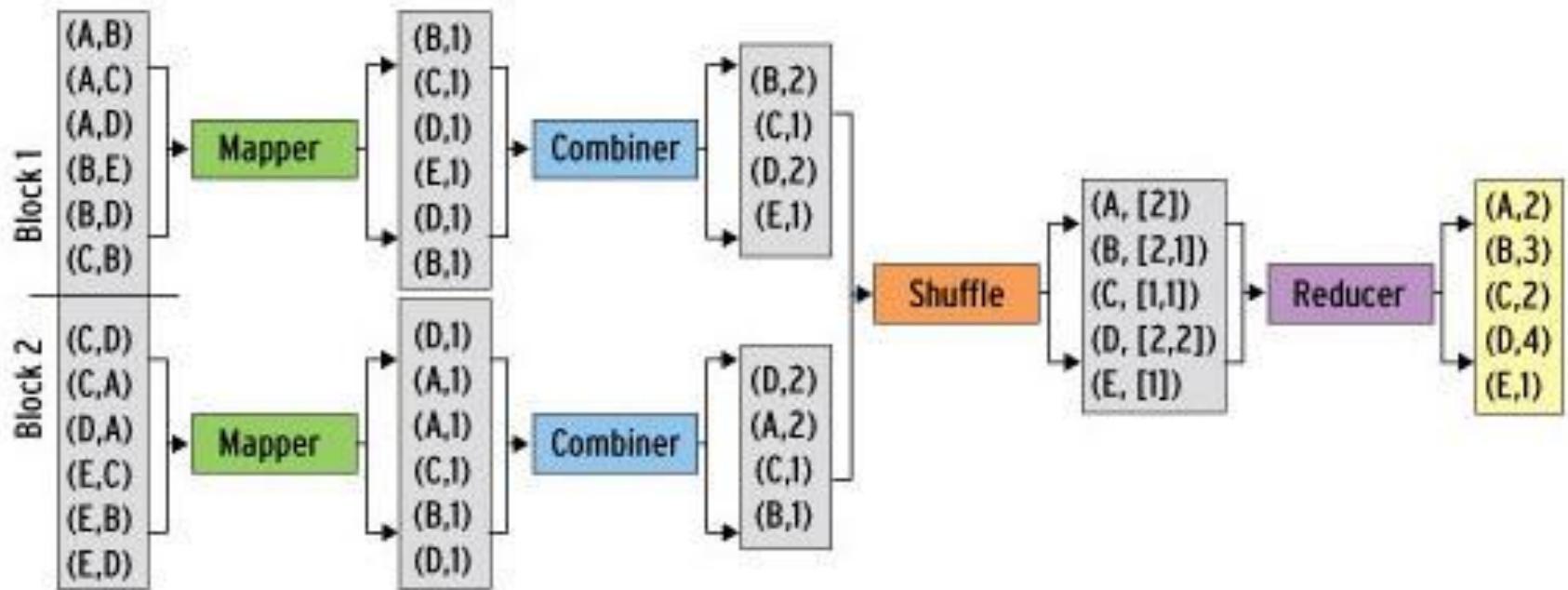
Example: Word Count, Combiner

Task: Calculate **word frequency** in a set of documents

```
combine(keyValuePairs):
    """ keyValuePairs: a list counts """
    result = {}
    for k, v in keyValuePairs:
        result[k] = result.get(k, 0) + v
    for k, v in result:
        emit(k, v);
```



Word Count with Combiner



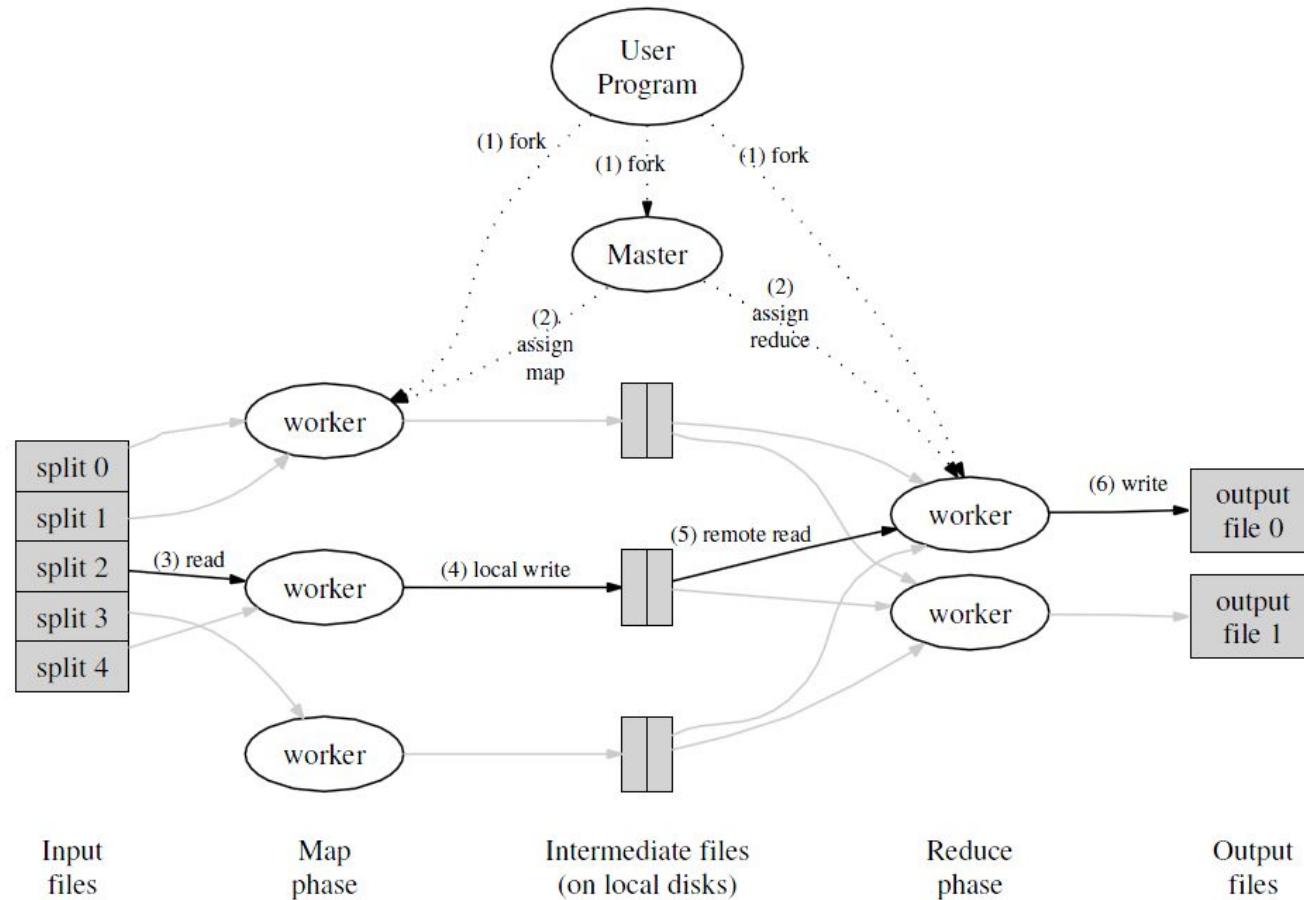


MapReduce Framework

- ❖ MapReduce **framework** takes care about
 - **Distribution** and parallelizing of the computation
 - **Monitoring** of the whole distributed task
 - The **grouping** phase
 - putting together intermediate results
 - **Recovering** from any failures
- ❖ User must define **only** map & reduce **functions**
 - but can define also other additional functions



MapReduce Framework





MapReduce Framework: Details

1. **Input reader** (function)
 - defines how to **read data** from underlying storage
2. **Map** (phase)
 - **master** node **prepares** M data **splits** and M **idle** Map tasks
 - pass individual splits to the Map tasks that run on **workers**
 - these map tasks are then **running**
 - when a task is **finished**, its intermediate results are stored
3. **Combiner** (function, optional)
 - **combine** local intermediate output from the Map phase



MapReduce Framework: Details

4. Partition (function)
 - to **partition** intermediate results for individual **Reducers**
5. Comparator (function)
 - sort and **group** the input for each Reducer
6. Reduce (phase)
 - **master** node creates R **idle** Reduce tasks on **workers**
 - **Partition** function **defines** a data **batch** for each reducer
 - each Reduce task uses **Comparator** to create **key-values pairs**
 - function **Reduce** is **applied** on each key-values pair
7. Output writer (function)
 - defines how the **output** key-value pairs are **written out**



MapReduce: Example II

Task: Calculate **graph** of web links

- ❖ what pages reference () each page (backlinks)

```
map(url, html):
    """ url: web page URL
        html: HTML text of the page """
    for tag, contents in html:
        if tag.type == 'a':
            emitIntermediate(tag.href, url)

reduce(key, values):
    """ key: target URLs
        values: a list of source URLs """
    emit(key, values)
```



Example II: Result

Input: (page_URL, HTML_code)

```
("http://cnn.com", "<html>...<a href=\"http://cnn.com\">link</a>...</html>")  
("http://ihnéd.cz", "<html>...<a href=\"http://cnn.com\">link</a>...</html>")  
("http://idnes.cz",  
  "<html>... <a href=\"http://cnn.com\">x</a>...  
    <a href=\"http://ihnéd.cz\">y</a>...  
    <a href=\"http://idnes.cz\">z</a>... </html>")
```

Intermediate output after Map phase:

```
("http://cnn.com", "http://cnn.com")  
("http://cnn.com", "http://ihnéd.cz")  
("http://cnn.com", "http://idnes.cz")  
("http://ihnéd.cz", "http://idnes.cz")  
("http://idnes.cz", "http://idnes.cz")
```

Intermediate result after shuffle phase (the same as output after Reduce phase):

```
("http://cnn.com", ["http://cnn.com", "http://ihnéd.cz", "http://idnes.cz"] )  
("http://ihnéd.cz", [ "http://idnes.cz" ] )  
("http://idnes.cz", [ "http://idnes.cz" ] )
```



MapReduce: Example III

Task: What are the **lengths** of words in the input text

- ❖ output = **how many** words are in the text for **each length**

```
map(key, text):  
    """ key: document name (ignored)  
        text: content of document (words) """  
    for w in text.split(' '):  
        emitIntermediate(length(w), 1)
```



```
reduce(key, values):  
    """ key: a length  
        values: a list of counts """  
    result = 0;  
    for v in values:  
        result += v  
    emit(key, result)
```



MapReduce: Features

- ❖ MapReduce uses a “**shared nothing**” architecture
 - Nodes operate **independently**,
 - shares no memory
 - shares no disk
 - Common feature of many NoSQL systems

- ❖ Data **partitioned** and **replicated** over many nodes
 - Pro: **Large** number of **read/write** operations per second
 - Con: **Coordination** problem – which nodes have my data, and when?



Applicability of MapReduce

- ❖ MR is always applicable if the problem is trivially **parallelizable**
- ❖ Two problems:
 1. The programming **model** is limited
(only two phases with a **given schema**)
 2. There is **no data model** - it works only on “data chunks”
- ❖ Google’s **answer** to the 2nd problem was **BigTable**
 - The first **column-family** system (2005)
 - Subsequent systems: HBase (over Hadoop), Cassandra,...



Apache Hadoop

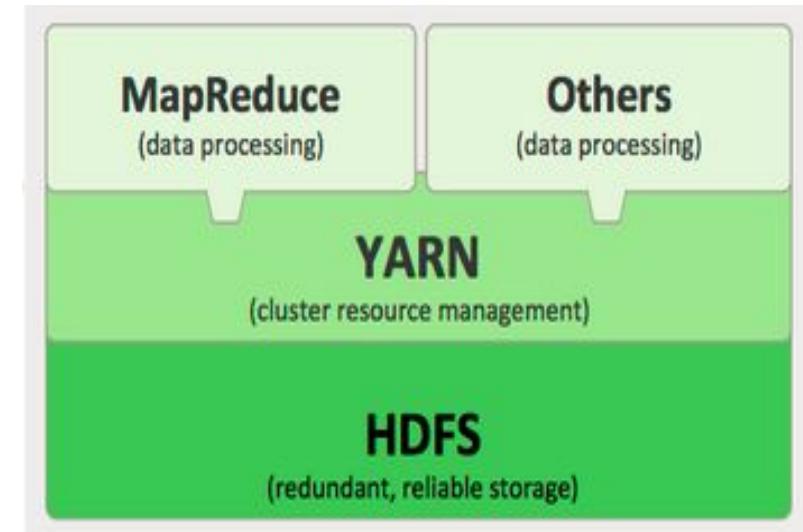
- ❖ Open-source MapReduce framework
 - Implemented in Java
 - Named for author's (Doug Cutting) son's yellow toy elephant
- ❖ Able to run applications on large clusters of commodity hardware
 - Multi-terabyte data-sets
 - Thousands of nodes
- ❖ A reimplementation and redesign of Google's MapReduce and Google File System





Hadoop: Modules

- ❖ Hadoop Common
 - Common support functions for other Hadoop modules
- ❖ Hadoop Distributed File System (HDFS)
 - Distributed file system
 - High-throughput access to application data
- ❖ Hadoop YARN
 - Job scheduling and cluster resource management
- ❖ Hadoop MapReduce
 - YARN-based system for parallel data processing



source: <https://goo.gl/NPuuJr>



HDFS: Data Characteristics

- ❖ Assumes:
 - **Streaming** data access
 - files are read sequentially from the beginning to end
 - **Batch processing** rather than interactive user access
- ❖ Very large data sets and files
- ❖ **Write-once / read-many**
 - A file once created does not change often
 - This assumption simplifies consistency
- ❖ Typical **applications** for this model:
MapReduce, web-crawlers, data warehouses, ...

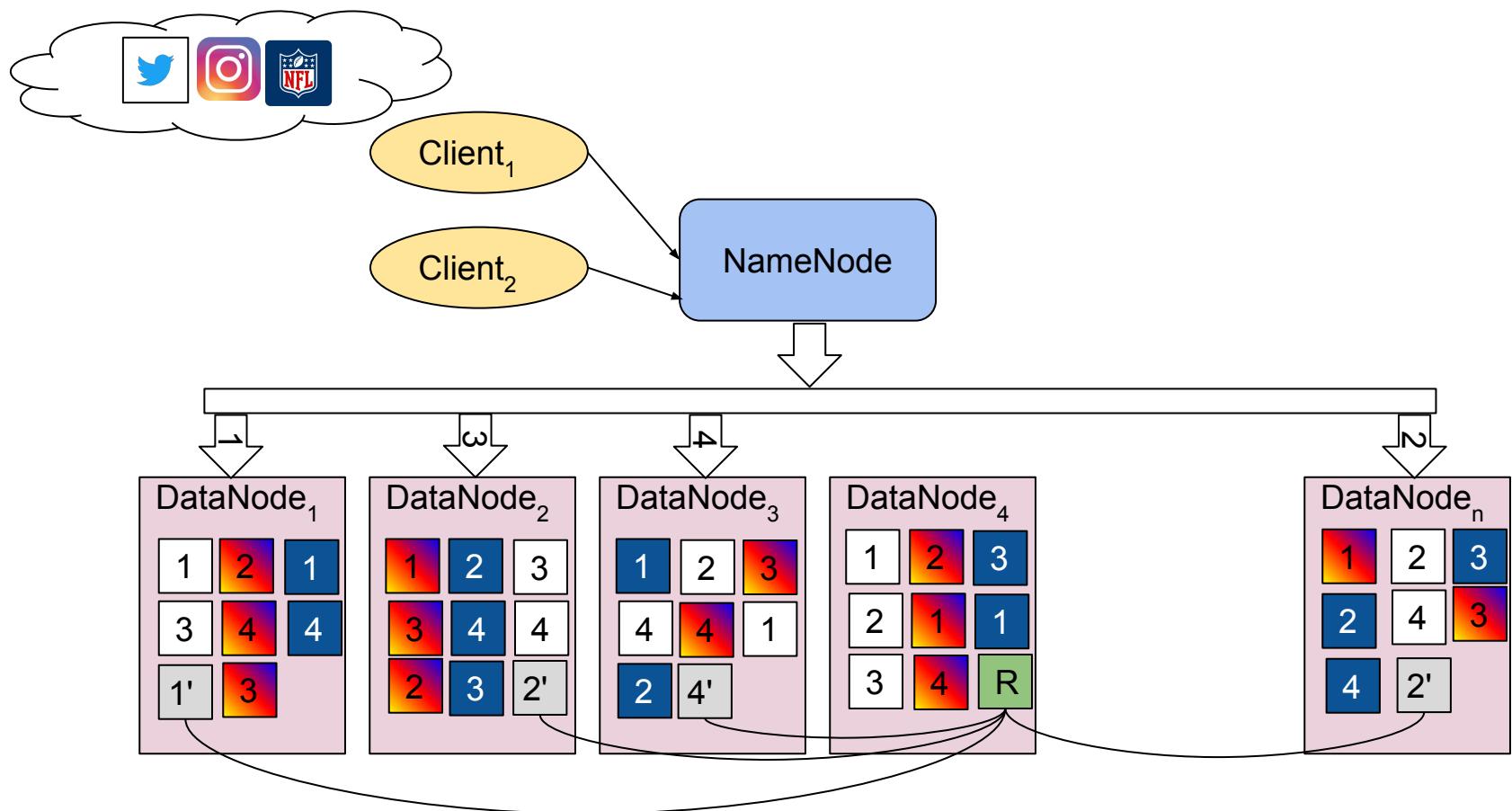


HDFS: Basic Components

- ❖ Master/slave architecture
- ❖ HDFS exposes a file system namespace
 - Files are internally split into blocks and distributed over servers called "DataNodes"
 - Blocks are relatively large (64 MB by default)
- ❖ NameNode - master server
 - Manages the file system namespace
 - Opening/closing/renaming files and directories
 - Arbitrates file access
 - Determines mapping of blocks to DataNodes
- ❖ DataNode - manages file blocks
 - Block read/write/creation/deletion/replication
 - Usually one per physical node



HDFS: Schema





HDFS: *NameNode*

- ❖ **NameNode** has a structure called **FsImage**
 - Entire **file system** namespace + mapping of **blocks** to files + file system properties
 - Stored in a file in NameNode's local file system
 - Designed to be **compact**
 - Loaded in NameNode's memory (4 GB of RAM is sufficient)
- ❖ **NameNode** uses a **transaction log** called **EditLog**
 - to **record every change** to the file system's meta data
 - E.g., creating a new file, change in replication factor of a file, ..
 - **EditLog** is stored in the NameNode's local file system



HDFS: *DataNode*

- ❖ Stores **data blocks** as files on its local file system
 - Each HDFS block is a **separate file**
 - Has **no knowledge** about HDFS file system

- ❖ When the DataNode **starts up**:
 - It **generates** a list of all HDFS blocks = **BlockReport**
 - It sends the report to NameNode

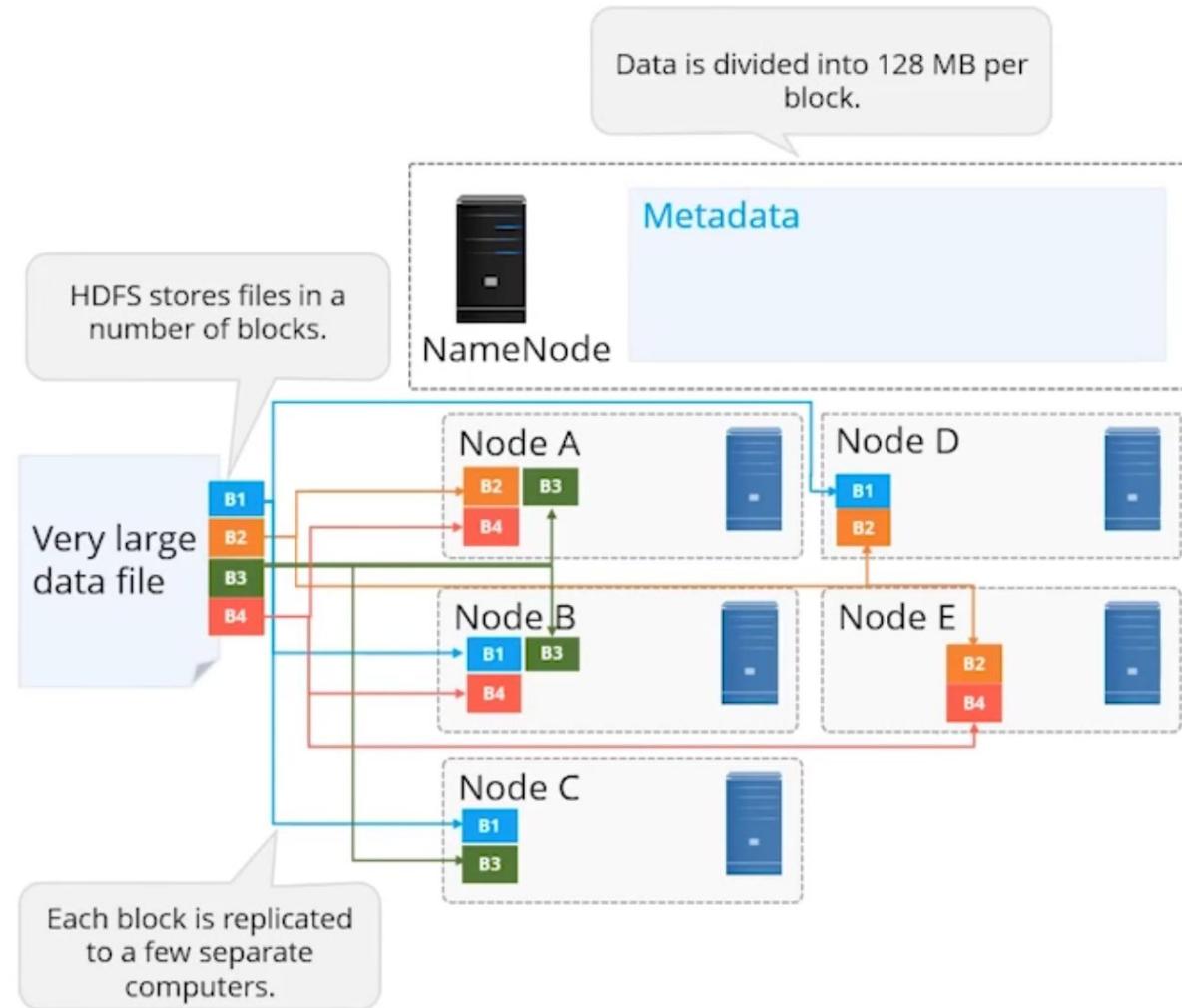


HDFS: Blocks & Replication

- ❖ HDFS can **store** very **large files** across a cluster
 - Each **file** is a sequence of **blocks**
 - All blocks in the file are of the same size
 - Except the last one
 - Block size is configurable per file (default 128MB)
 - Use of large files promotes high I/O throughput
 - **Blocks are replicated** for fault tolerance
 - Number of replicas is configurable per file
- ❖ NameNode receives **HeartBeat** and **BlockReport** from each DataNode
 - **BlockReport:** **list of all blocks** on a DataNode



HDFS: Block Replication





HDFS: Reliability

- ❖ Primary objective: to store data **reliably** in case of:
 - NameNode failure
 - DataNode failure
 - Network partition
 - a subset of DataNodes can lose connectivity with NameNode
- ❖ In case of **absence of** a HeartBeat message
 - NameNode **marks** DataNodes without HeartBeat and does **not send** any I/O requests to them
 - The death of a DataNode typically results in **re-replication**

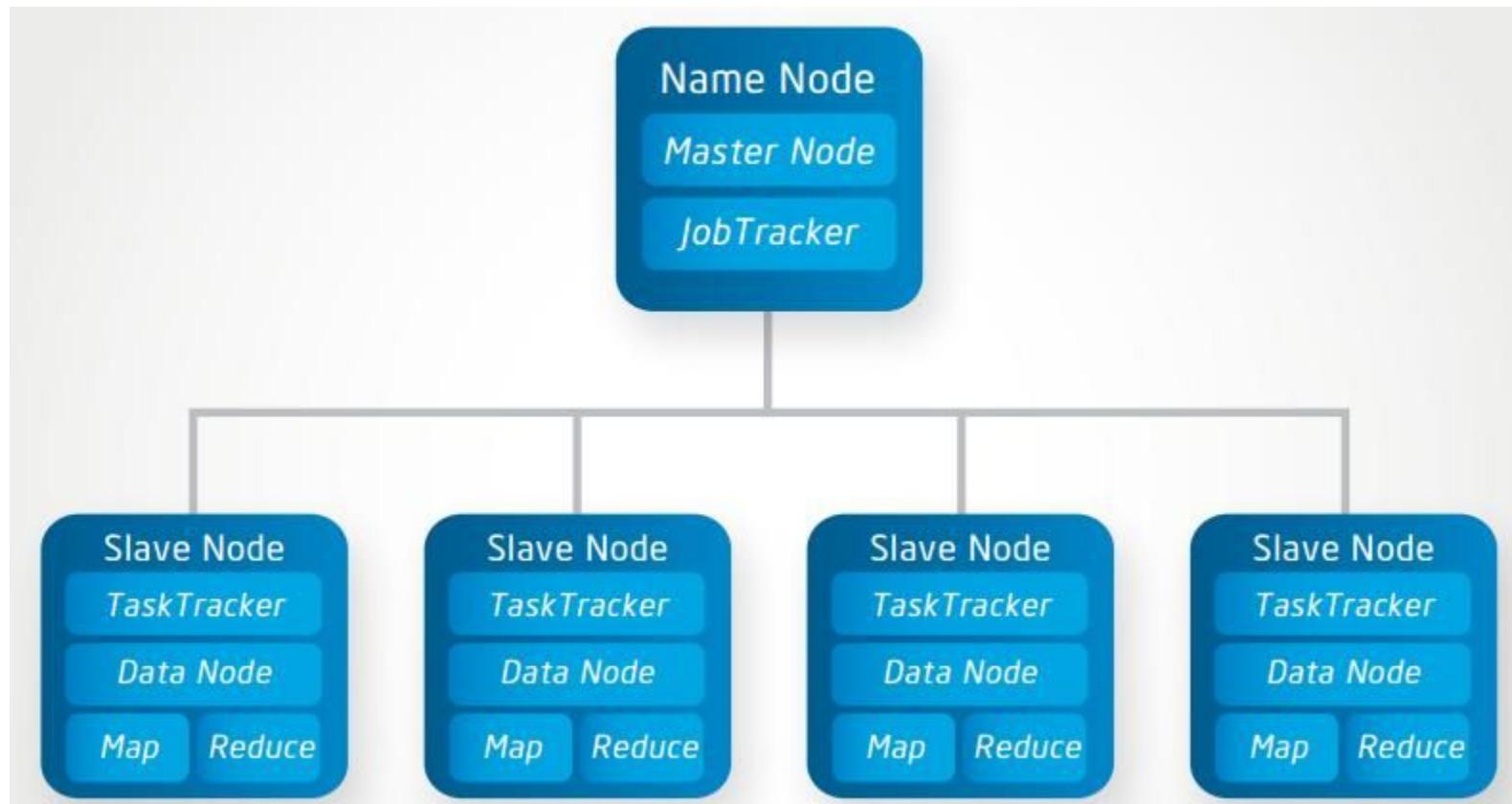


Hadoop: MapReduce

- ❖ Hadoop MapReduce **requires**:
 - Distributed file system (typically **HDFS**)
 - Engine that can distribute, coordinate, monitor and gather the results (typically **YARN**)
- ❖ Two main **components**:
 - **JobTracker** (master) = scheduler
 - tracks the whole MapReduce job
 - communicates with HDFS NameNode to run the task close to the data
 - **TaskTracker** (slave on each node) – is assigned a Map or a Reduce task (or other operations)
 - Each **task** runs in its own **JVM**

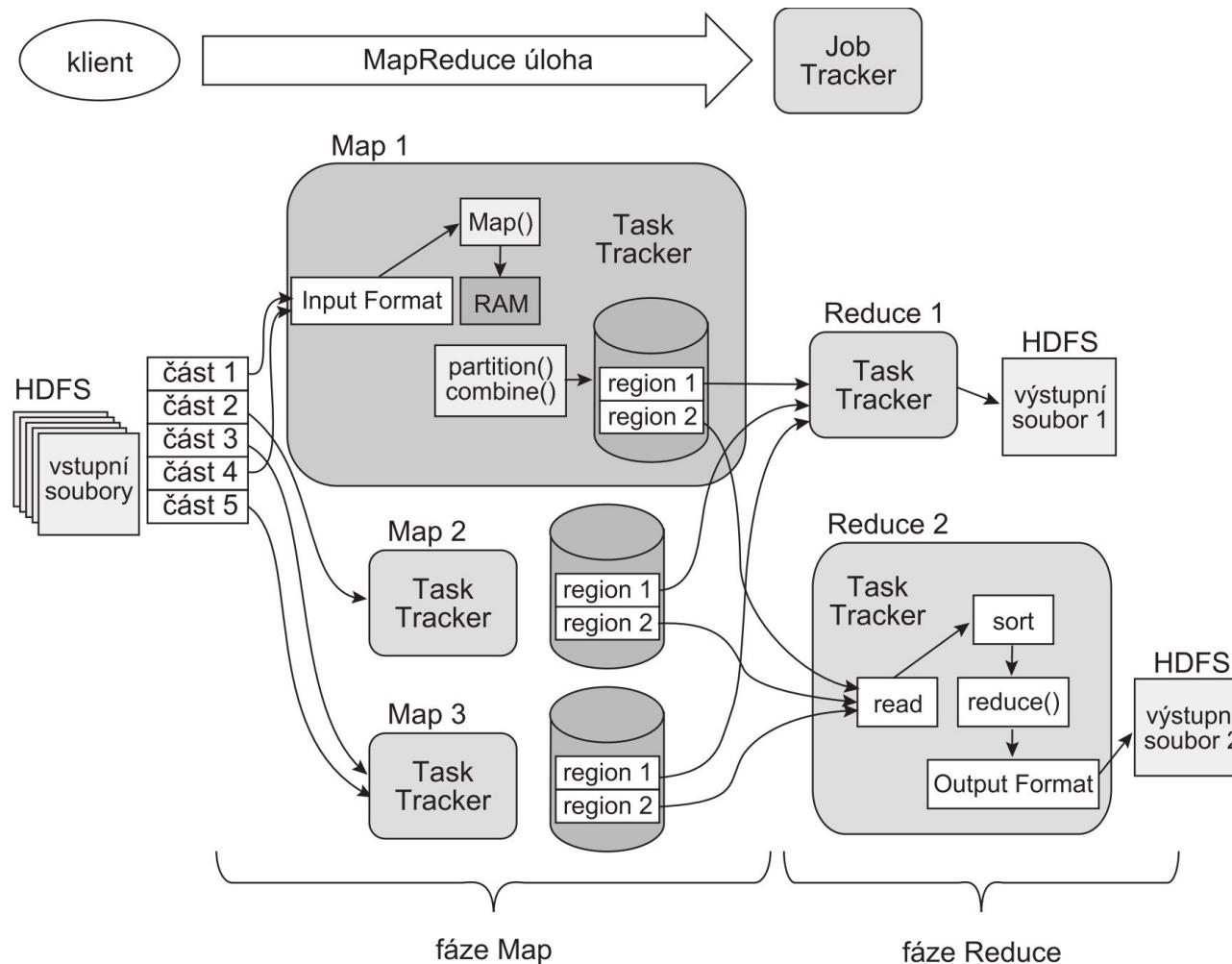


Hadoop HDFS + MapReduce





Hadoop MapReduce: Schema





Hadoop MR: WordCount Example (1)



```
public class Map
    extends Mapper<LongWritable, Text, Text, IntWritable> {

private final static IntWritable one = new IntWritable(1);
private final Text word = new Text();

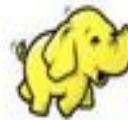
@Override protected void map(LongWritable key, Text value,
    Context context) throws ... {
    String string = value.toString()
    StringTokenizer tokenizer = new StringTokenizer(string);
    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
    }
}
}
```



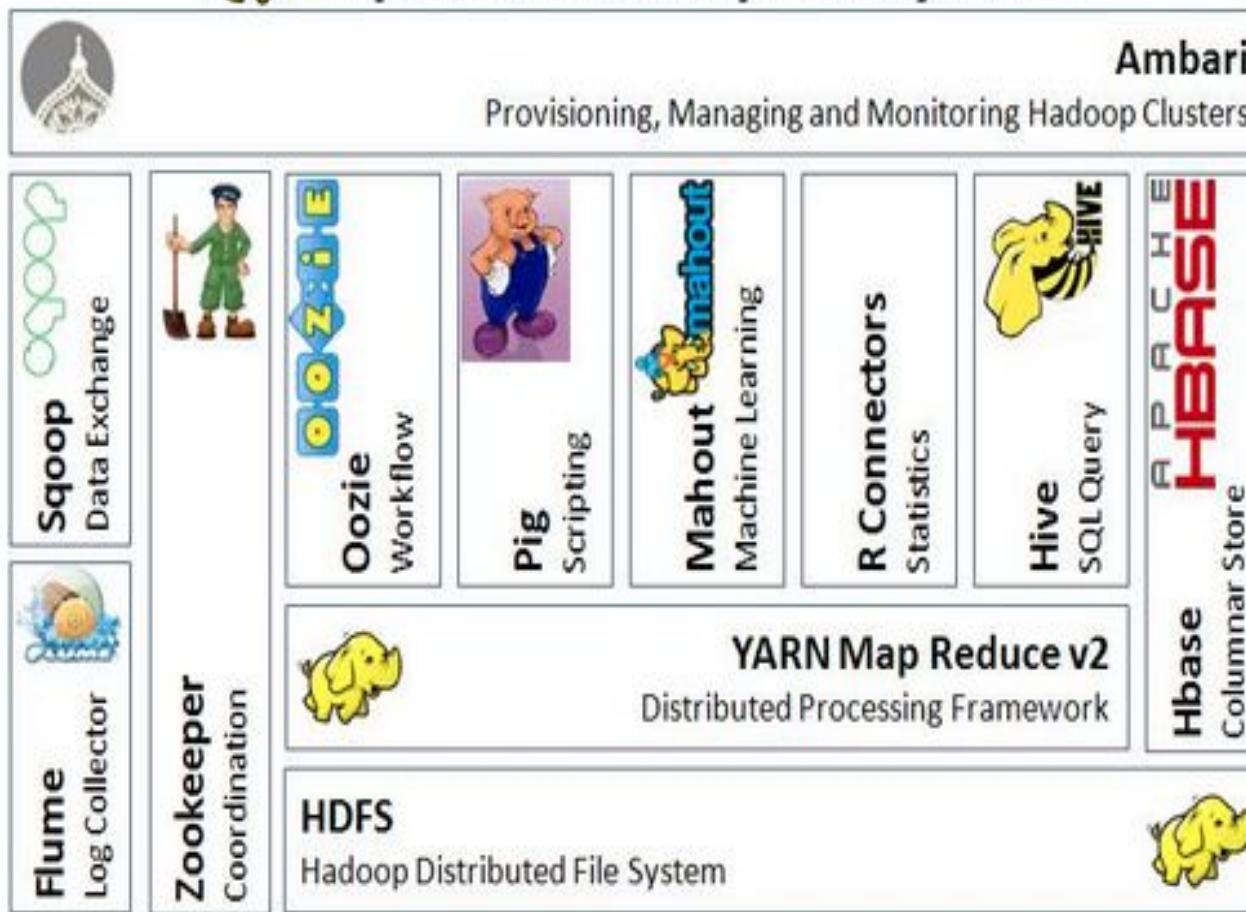
Hadoop MR: WordCount Example (2)

```
public class Reduce
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce (Text key, Iterable<IntWritable> values,
        Context context) throws ... {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```



Apache Hadoop Ecosystem





Next time

- A shallow dive into the Hadoop Eco system
- Primarily, Pig and Hive





Programming in Hadoop with Pig and Hive





Hadoop Review

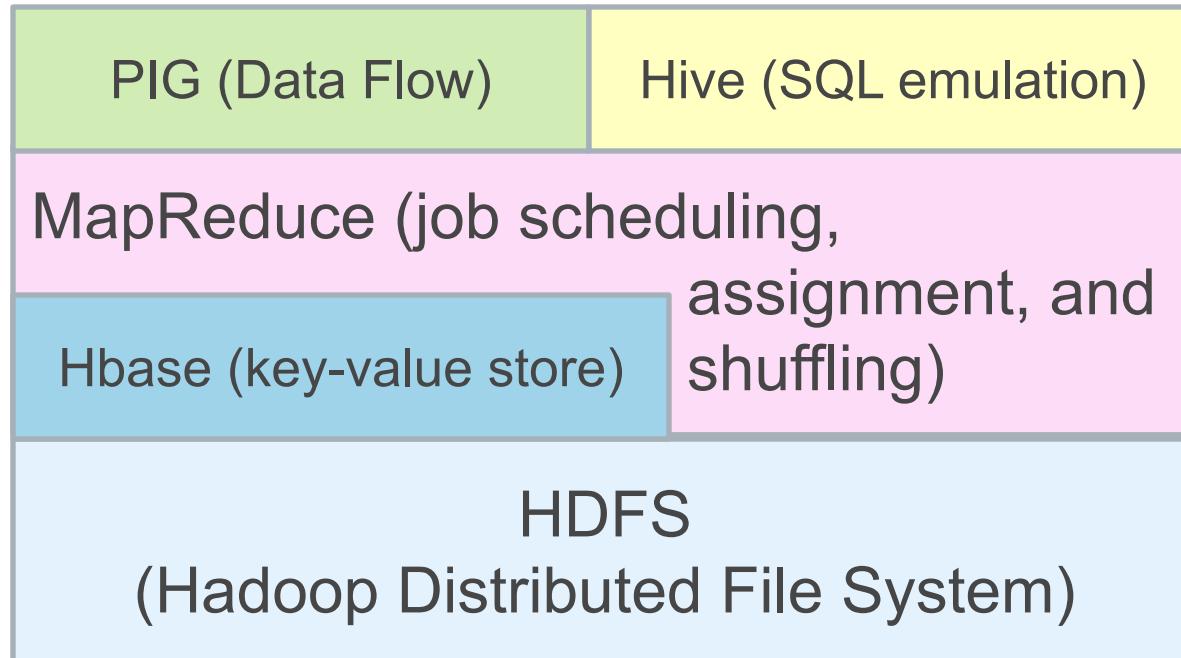
- Hadoop is a open-source reimplementation of
 - A distributed file system
 - A map-reduce processing framework
- Inspired by Google's description of the technologies underpinning their search engine
- It is a layer-cake of APIs, written mostly in Java, that one can use to write large, distributed, and scalable applications to search and process large datasets



Hadoop Layer Cake

While Hadoop has many advantages, it is not intuitive to translate every data exploration/manipulation/searching task into a series of map-reduce operations.

Higher-level languages were needed.





High-level Hadoop Interfaces

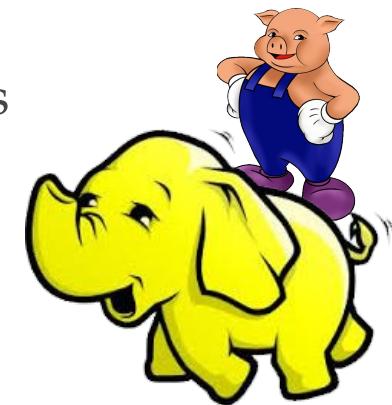
- **PIG** – A scripting language for transforming big data
 - Useful for “cleaning” and “normalizing” data
 - Three parts:
 - Pig Latin – The scripting language
 - Grunt – A interactive shell
 - Piggybank – A repository of Pig extensions
 - Deferred execution model
- **Hive** – A SQL-inspired query-oriented language
 - Imposes structure, in the form of schemas, on Hadoop data
 - Creates “data warehouse” layers





Pig Latin's data model

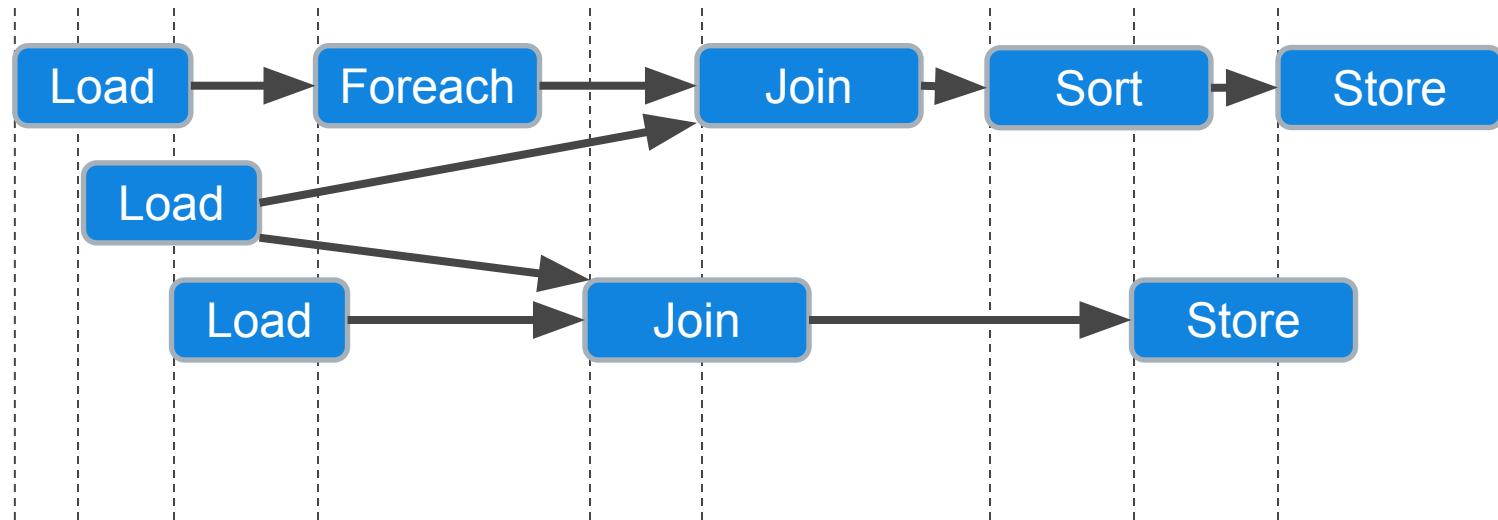
- PIG – A dataflow scripting language
 - Automatically translated to a series of Map-Reduce jobs that are run on Hadoop
 - It requires no meta-data or schema
 - It is extensible, via user-defined functions (UDFs) written in Java or other languages (C, Python, etc.)
 - Provides run-time and debugging environments
 - A language specifically designed for data manipulations and analysis
 - Supports join, sort, filter, etc.
 - Automatically partitions large operations into smaller jobs and chains them together





Pig Latin scripts describe dataflows

- Every Pig Latin script describes one or more flows of data through a series of operations that can be processed in parallel (i.e. the next one can start before the ones providing inputs to it finish).
- Dataflows are Directed Acyclic Graphs (DAGS)
- Ordering and Scheduling is deferred until a node requires data





Pig Latin Processing

- Pig Latin script are processed line by line
 - Syntax and References are checked
 - Valid statements are added to a logical plan
 - Execution is deferred until either a DUMP or STORE statement is reached
 - Reused intermediate results are mapped to a common node



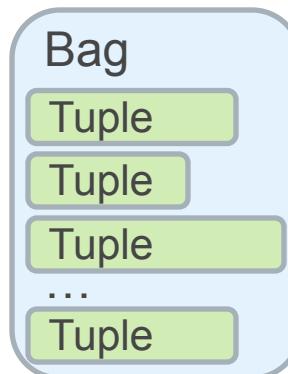
/An HDFS filename, don't worry
about where it really is.

```
grunt> roster = LOAD "comp521/NFLrosters" USING PigStorage(',');
grunt> RecentQBs = FILTER roster BY $5='QB' AND $1>2000;
grunt> DUMP RecentQBs;
```



Pig Relations

- Pig variables are bags of tuples
 - Fields – data items
 - Tuples – a vector of fields
 - Bags – a collection of unordered tuples
 - Unlike *Relations* in relational databases the tuples in a *Pig bag*, need not have the same number of fields, or the same types
- Pig also supports Maps
 - Maps – a dictionary of name-value pairs





Pig Latin Examples

- Pig scripts are easy to read

 With AS we define an "on-the-fly" data schema

```
roster = LOAD "comp521/NFLrosters" USING PigStorage(',') AS  
    (team:chararray, year:int, jersey:int, name:chararray,  
     position:chararray, starts:int, games:int);  
recentQBs = FILTER roster BY position='QB' AND year>2000 AND starts > 0;  
Groups = GROUP recentQBs BY name;  
STORE Groups INTO "recentQBTable";
```

- FOREACH to specify processing steps for all tuples in a bagexample

```
e1 = LOAD "input/Employees" USING PigStorage(',') AS  
    (name:chararray, age:int, zip:int, salary:double);  
f = FOREACH e1 GENERATE age, salary; -- a projection  
DESCRIBE f; -- gives the schema of relation f  
DUMP f;
```



More Pig Latin Examples

- ORDER

```
emp = LOAD "input/Employees" USING PigStorage(',') AS  
    (name:chararray, age:int, zip:int, salary:double);  
sorted = ORDER emp BY salary;
```

- LIMIT, SAMPLE

```
emp = LOAD "input/Employees" USING PigStorage(',') AS  
    (name:chararray, age:int, zip:int, salary:double);  
agegroup = GROUP emp BY age;  
shortlist = LIMIT agegroup 100;
```

- JOIN

```
emp = LOAD "input/Employees" USING PigStorage(',') AS  
    (name:chararray, age:int, zip:int, salary:double);  
pbk = LOAD "input/Phonebook" USING PigStorage(',') AS  
    (name:chararray, phone:chararray);  
contact = JOIN emp BY name, pbk BY name;  
DESCRIBE contact;  
DUMP contact;
```



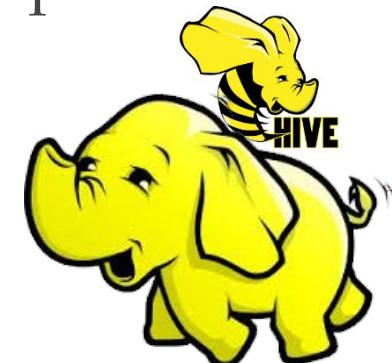
Hive Query Language

- Hive is an alternative/complement to Pig
 - Developed by Facebook around 2007
 - Hive is a "*SQL-like*" Query language
 - It imposes "Structure" on "Unstructured" data
 - Needs a predefined schema definition
 - It is also extensible, via user-defined functions (UDFs) written in Java or other languages (C, Python, etc.)
- Hive isn't a relational database
 - No transactions, no isolation, no consistency promises
 - Searches and processes Hadoop data stores
 - Not suitable for real-time queries and row-level updates
 - Generally much higher latency than a DBMS, but higher performance
 - Best for batch jobs over large "immutable" data



Hive Usage

- Hive is best used to perform analyses and summaries over large data sets
- Hive requires a meta-store to keep information about virtual tables
- It evaluates query plans, selects the most promising one, and then evaluates it using a series of map-reduce functions
- Hive is best used to *answer a single instance of a specific question* whereas Pig is best used to accomplish *frequent reorganization, combining, and reformatting tasks*





Hive Interface

- Hive is similar to SQL-92
- Based on familiar database concepts, tables, rows, columns, and schemas
- Makes "Big Data" appear as tables on the fly
- Like Pig, Hive has a command-line shell

```
$ hive  
hive>
```

- Or it can execute scripts

```
$ hive -f myquery.hive
```

- There are also GUIs



Defining Hive Tables

- A Hive table consists of
 - Data linked to a file or multiple files in an HDFS
 - Schema stored as mapping of the data to a set of columns with types
- Schema and Data are separated
 - Allows multiple schemas on the same data

```
$ hive
hive> CREATE TABLE Roster (
    team string,
    year int,
    jersey string,
    player string,
    position string,
    starts int,
    games int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```



Operations on Hive Tables

Table Operation	Command Syntax
See current tables	hive> SHOW TABLES;
Check schema	hive> DESCRIBE Roster;
Change table name	hive> ALTER TABLE Roster RENAME TO PlayedFor;
Add a column	hive> ALTER TABLE Roster ADD COLUMNS (pid, int);
Drop a partition	hive> ALTER TABLE Roster DROP PARTITION (team='dallas-cowboys');



Loading Hive Tables

- Use LOAD DATA to import data into a HIVE table

```
$ hive  
hive> LOAD DATA LOCAL INPATH 'comp521/NFLrosters'  
      INTO TABLE Roster;
```

- No files are modified by Hive, the schema simply imposes structure on the file as it is read
- You can use the keyword OVERWRITE to modify previous loaded files

```
hive> LOAD DATA INPATH 'comp521/NFLrosters'  
hive>      OVERWRITE INTO TABLE Rosters;  
hive> INSERT INTO recentQBs  
hive>      SELECT * FROM Rosters  
hive>      WHERE position = 'QB' AND year > 2000  
hive>      AND starts > 0;
```



Hive Queries

- SELECT

```
$ hive  
hive> SELECT * FROM recentQBs WHERE name = "*Brady";
```

- Supports the following:
 - WHERE clause
 - UNION ALL
 - DISTINCT
 - GROUP BY and HAVING
 - LIMIT
 - JOIN,
 - LEFT OUTER JOIN, RIGHT OUTER JOIN, OUTER JOIN
 - Returned rows are random, and may vary between calls



Hive Query Examples

```
hive> SELECT * FROM customers;
hive> SELECT COUNT(*) FROM customers;
hive>
hive> SELECT first, last, address, zip FROM customers
hive> WHERE orderID > 0
hive> GROUP BY zip;
hive>
hive> SELECT customers.*, orders.*
hive> FROM customers JOIN orders
hive>   ON (customers.customerID = orders.customerID);
hive>
hive> SELECT customers.*, orders.*
hive> FROM customers LEFT OUTER JOIN orders
hive>   ON (customers.customerID = orders.customerID);
```

- If you understand SQL, you should be able to follow
- Note: These are queries, not transactions
- The data's state could change between and within a query



Hive Subqueries

- Hive allows subqueries only within FROM clauses

```
hive> SELECT sid, mid, total FROM
hive>      (SELECT sid, mid, refCnt + altCnt AS total
hive>          FROM genotype) gtypeTotals
hive> WHERE total > 20;
```

- Subqueries are generally materialized (computed and saved as hive tables)
- You MUST to include a name for the subquery result table
- The columns of a subquery's SELECT list are available to the outer query



Sorting in Hive

- Hive supports ORDER BY, but its result differs from SQL's
 - Only one Reduce step is applied and partial results are broadcast and combined
 - No need for any intermediate files
 - This allows optimization to a single MapReduce step
- Hive also supports SORT BY with multiple fields
 - Produces a "total ordering" of all results
 - Might require multiple MapReduce operations
 - Might materialize several intermediate tables



Summary

- There are two primary "high-level" programming languages for Hadoop-- Pig and Hive
- *Pig* is a "scripting language" that excels in specifying a processing pipeline that is automatically parallelized into Map-Reduce operations
 - Deferred execution allows for optimizations in scheduling Map-Reduce operations
 - Good for general data manipulation and cleaning
- *Hive* is a "query language" that borrows heavily from SQL, good for searching and summarizing data
 - Requires the specification of an "external" schema
 - Often materializes many more intermediate results than a DBMS would



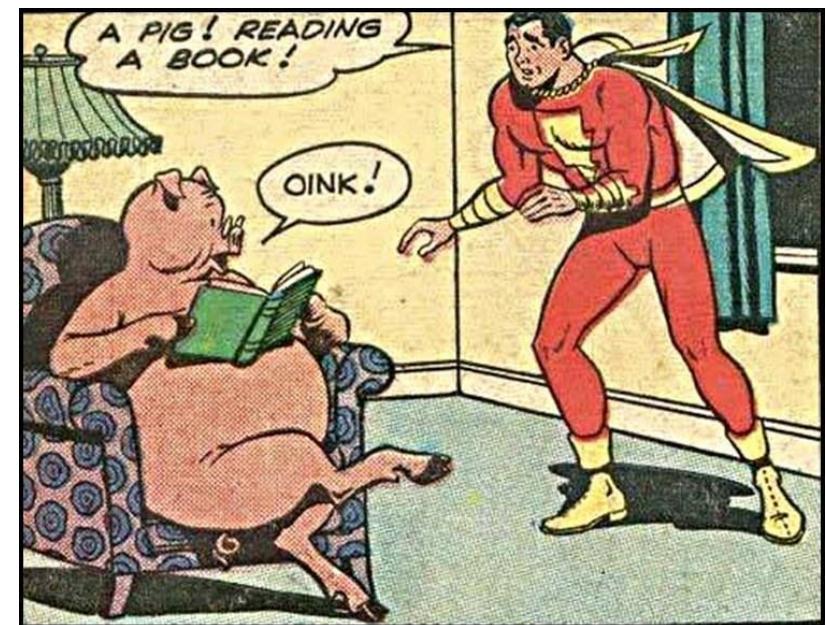


NoSQL

Document Databases

Problem Set #5 is
due on Thursday

Problem Set #6 is coming,
but getting simpler
every day it is delayed





NoSQL Databases and Data Types

1. Key-value stores:

- Can store **any** (text or binary) **data**
 - often, if using JSON data, additional functionality is available

2. Document databases

- **Structured text** data - Hierarchical tree data structures
 - typically JSON, XML

3. Columnar stores

- Rows that have **many columns** associated with a **row key**
 - can be written as JSON



Unstructured Data Formats

❖ **Binary Data**

- often, we want to store **objects** (class instances)
- objects can be binary **serialized** (**marshalled**)
 - and kept in a key-value store
- there are several popular **serialization formats**
 - Protocol Buffers, Apache Thrift

❖ **Structured Text Data**

- JSON, BSON (Binary JSON)
 - **JSON** is currently **number one** data format used on the **Web**
- XML: eXtensible Markup Language
- RDF: Resource Description Framework



JSON: Basic Information

- ❖ Text-based open **standard** for data interchange
 - Serializing and transmitting structured data
- ❖ JSON = JavaScript Object Notation
 - Originally specified by Douglas Crockford in 2001
 - Derived **from JavaScript** scripting language
 - Uses conventions of the C-family of languages
- ❖ Filename: *.json
- ❖ Internet media (MIME) type: **application/json**
- ❖ Language independent



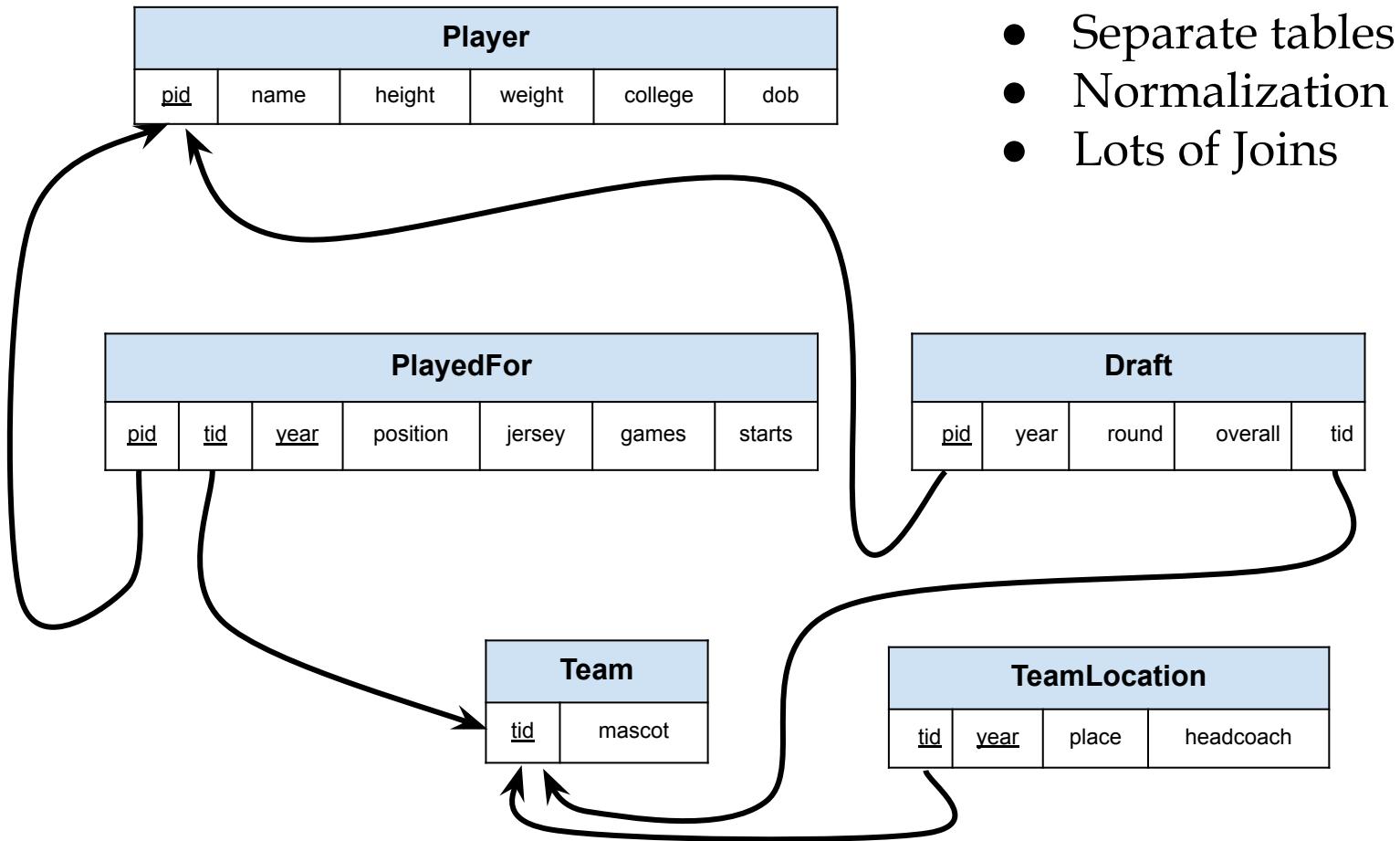
JSON:Example



```
[{ 'pid': 28352, 'name': 'Brandin Cooks',
  'height': '5-10', 'weight': '189',
  'birthdate': '1993-09-25', 'college': 'Oregon St',
  'draft': {'team': 'New Orleans Saints', 'round': '1', 'order': 20, 'year': 2014 },
  'roster': [
    {'year': 2014, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 10, 'starts': 7 },
    {'year': 2015, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 16, 'starts': 12 },
    {'year': 2016, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 16, 'starts': 12 },
    {'year': 2017, 'team': 'New England Patriots', 'position': 'WR', 'jersey': '14', 'games': 16, 'starts': 15 },
    {'year': 2018, 'team': 'Los Angeles Rams', 'position': 'WR', 'jersey': '12', 'games': 16, 'starts': 16 }]
},
{ 'pid': 22721, 'name': 'Tom Brady',
  'height': '6-4', 'weight': '225',
  'birthdate': '1977-08-03', 'college': 'Michigan',
  'draft': {'team': 'New England Patriots', 'round': '6', 'order': 199, 'year': 2000},
  'roster': [
    {'year': 2000, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 1, 'starts': 0 },
    {'year': 2001, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 15, 'starts': 14 },
    {'year': 2002, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2003, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2004, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2005, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2006, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2007, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2008, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 1, 'starts': 1 },
    {'year': 2009, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2010, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2011, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2012, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2013, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2014, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2015, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2016, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 12, 'starts': 12 },
    {'year': 2017, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
    {'year': 2018, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 }]
```



Compared to a Relational DB

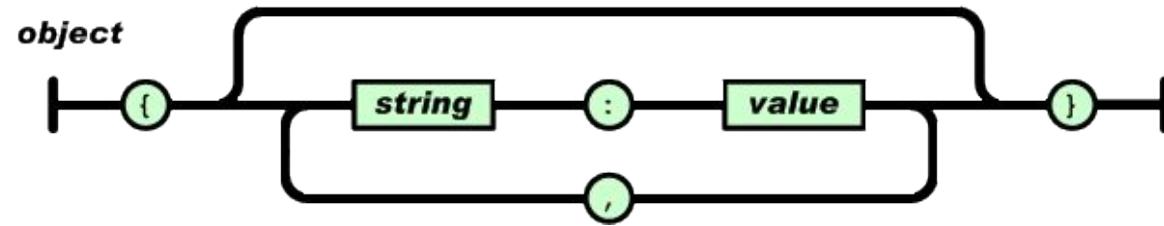


- Separate tables
- Normalization
- Lots of Joins

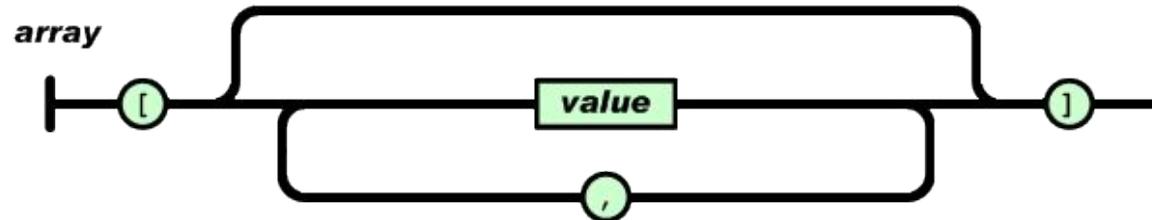


JSON: Data Types (1)

- ❖ **object** – an **unordered** set of **name+value** pairs
 - these pairs are called **properties** (members) of an object
 - syntax: { name: value, name: value, name: value, ... }



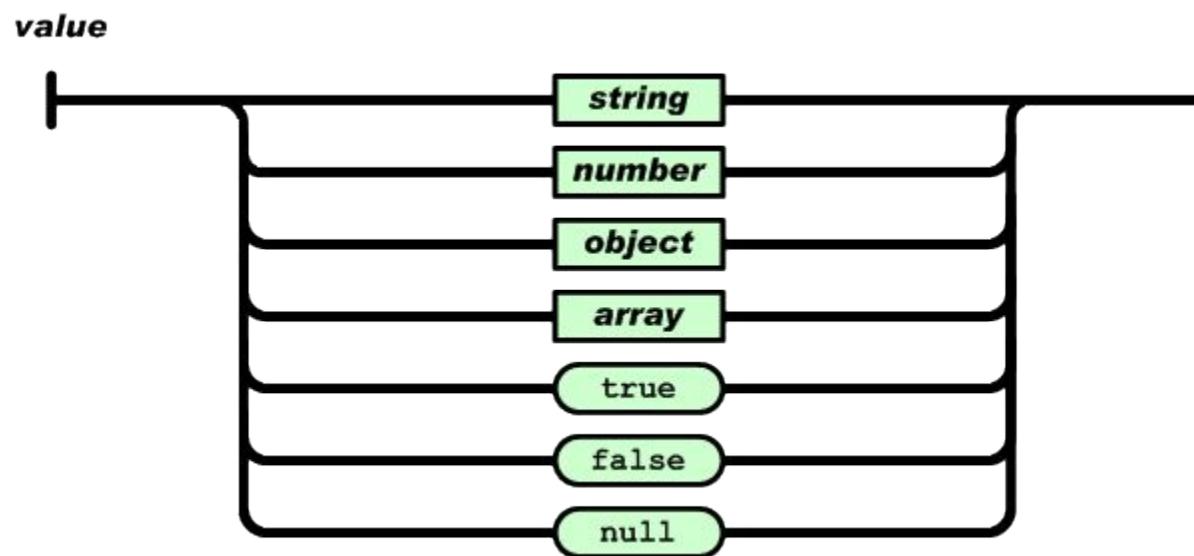
- ❖ **array** – an **ordered** collection of **values** (elements)
 - syntax: [comma-separated values]





JSON: Data Types (2)

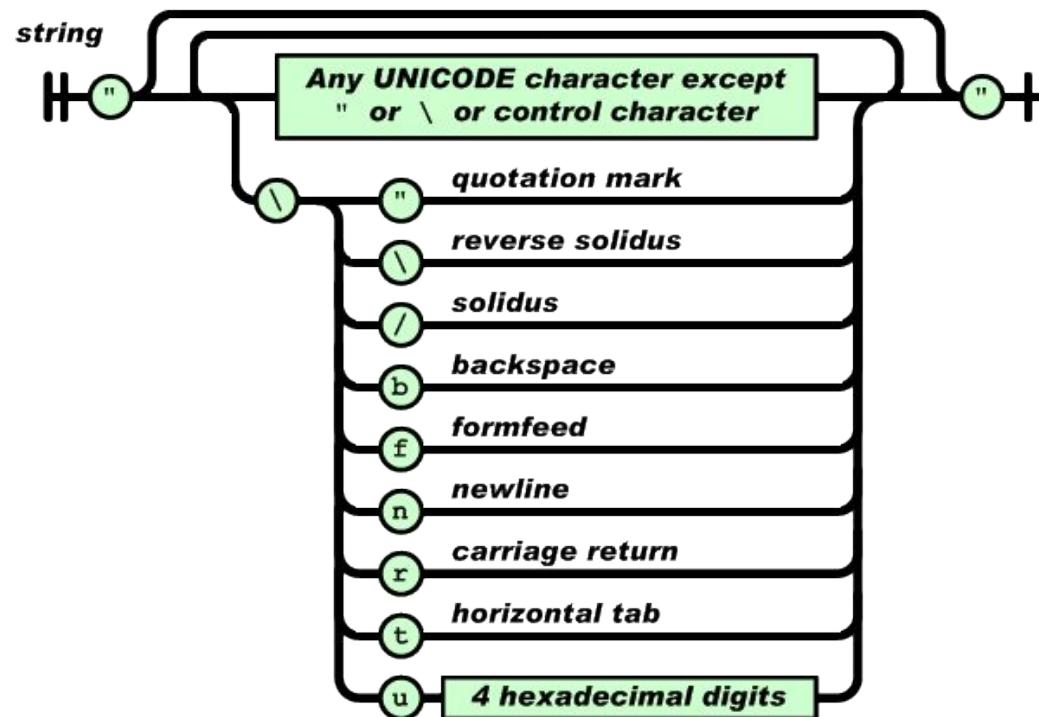
- ❖ **value** - string in double quotes / number / true or false (i.e., Boolean) / null / object / array





JSON: Data Types (3)

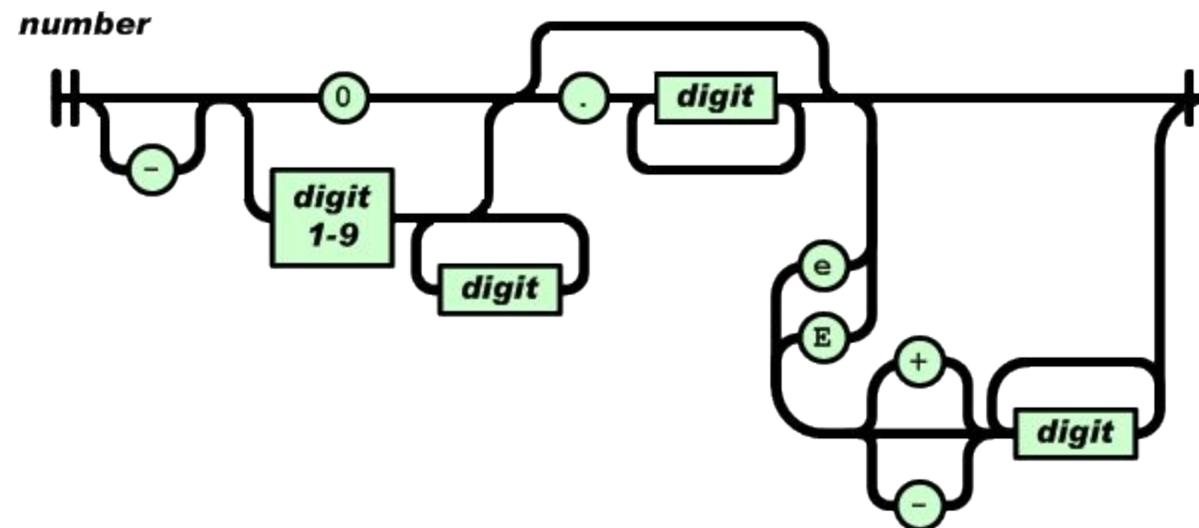
- ❖ **string** – sequence of zero or more **Unicode characters**, wrapped in double quotes
 - Backslash escaping





JSON: Data Types (4)

- ❖ **number** – like a C, Python, or Java number
 - Integer or float
 - Octal and hexadecimal formats are not used





JSON Properties

- ❖ There are **no comments** in JSON
 - Originally, there was but they were **removed** for **security**
- ❖ **No way** to specify **precision/size** of numbers
 - It depends on the **parser** and the programming language
- ❖ There **exists** a standard “**JSON Schema**”
 - A way to **specify** the **schema** of the data
 - Field **names**, field **types**, **required/optional** fields, etc.
 - JSON Schema is written in JSON, of course
 - see example below



JSON Schema: Example

```
{  
  "$schema": "http://json-schema.org/schema#",  
  "type": "object",  
  "properties": {  
    "conferences": {  
      "type": "array",  
      "items": {  
        "type": "object",  
        "properties": {  
          "name": { "type": "string" },  
          "start": { "type": "string", "format": "date" },  
          "end": { "type": "string", "format": "date" },  
          "web": { "type": "string" },  
          "price": { "type": "number" },  
          "currency": { "type": "string",  
            "enum": ["CZK", "USD", "EUR", "GBP"] },  
          "topics": {  
            "type": "array",  
            "items": {  
              "type": "string"  
            }  
          },  
          "...": {}  
        }  
      }  
    },  
    "venue": {  
      "type": "object",  
      "properties": {  
        "name": { "type": "string" },  
        "location": {  
          "type": "object",  
          "properties": {  
            "lat": { "type": "number" },  
            "lon": { "type": "number" }  
          }  
        }  
      }  
    },  
    "required": ["name"]  
  },  
  "required": ["name", "start", "end",  
    "web", "price", "topics"]  
}
```



Document with JSON Schema

```
{  
  "conferences": [  
    {  
      "name": "XML Prague 2015",  
      "start": "2015-02-13",  
      "end": "2015-02-15",  
      "web": "http://xmlprague.cz/",  
      "price": 120,  
      "currency": "EUR",  
      "topics": ["XML", "XSLT", "XQuery", "Big Data"],  
      "venue": {  
        "name": "VŠE Praha",  
        "location": {  
          "lat": 50.084291,  
          "lon": 14.441185  
        }  
      },  
      {  
        "name": "DATAKON 2014",  
        "start": "2014-09-25",  
        "end": "2014-09-29",  
        "web": "http://www.datakon.cz/",  
        "price": 290,  
        "currency": "EUR",  
        "topics": ["Big Data", "Linked Data", "Open Data"]  
      }  
    }  
  ]  
}
```



XML: Basic Information

- ❖ XML: eXtensible Markup Language
 - W3C standard (since 1996)
- ❖ Designed to be **both** human and machine **readable**
- ❖ example:

```
<?xml version="1.0"?>
<quiz>
<qanda seq="1">
<question>
Who was the forty-second
president of the U.S.A.?
</question>
<answer>
William Jefferson Clinton
</answer>
</qanda>
<!-- Note: We need to add
more questions later.-->
</quiz>
```

XML



XML: Features and Comparison

- ❖ Standard ways to specify XML document **schema**:
 - DTD, XML Schema, etc.
 - concept of Namespaces; XML editors (for given schema)
- ❖ Technologies for **parsing**: DOM, SAX
- ❖ Many associated **technologies**:
 - XPath, XQuery, XSLT (transformation)
- ❖ XML is good for **configurations, meta-data**, etc.
- ❖ **XML databases** are mature, not considered NoSQL
- ❖ Currently, **JSON** format **rules**:
 - **compact, easier** to write, meets most needs



NoSQL Document Databases

- ❖ Basic concept of data: *Document*
- ❖ Documents are **self-describing** pieces of data
 - **Hierarchical tree data structures**
 - Nested associative arrays (maps), collections, scalars
 - XML, JSON (JavaScript Object Notation), BSON, ...
- ❖ Documents in a **collection** should be “similar”
 - Their **schema** can **differ**
- ❖ Often: **Documents** stored as **values** of key-value
 - Key-value stores where the values are **examinable**
 - Building search **indexes** on various **keys/fields**

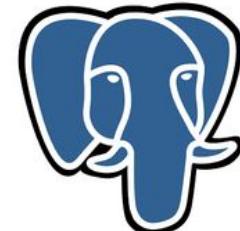


Why Document Databases

- ❖ XML and JSON are popular for **data exchange**
 - Recently mainly JSON
- ❖ Data stored in document DB can be used directly
- ❖ Databases often store **objects** from **memory**
 - Using **RDBMS**, we must do Object Relational Mapping (**ORM**)
 - ORM is relatively **demanding**
 - **JSON** is much **closer** to structure of **memory objects**
 - It was originally for JavaScript objects
 - **Object Document Mapping** (ODM) is faster



Document Databases





Example: MongoDB

- ❖ Initial release: 2009

- Written in C++
- Open-source
- Cross-platform

- ❖ JSON documents

- ❖ Basic **features**:

- High **performance** – many indexes
- High **availability** – replication + eventual consistency + automatic failover
- Automatic **scaling** – automatic sharding across the cluster
- **MapReduce** support

```
{  
    name: "sue",  
    age: 26,  
    status: "A",  
    groups: [ "news", "sports" ]  
}
```

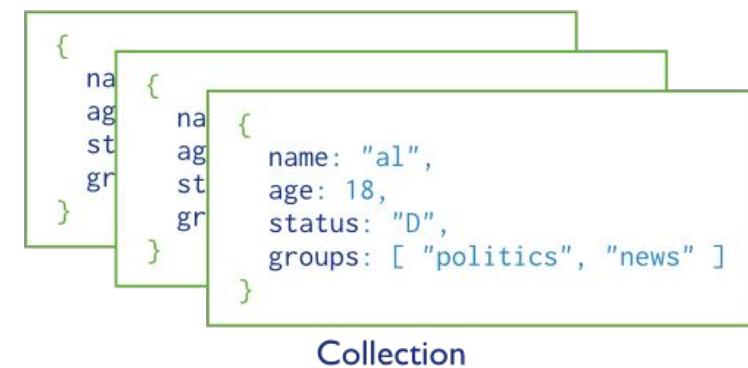
← field: value
← field: value
← field: value
← field: value



MongoDB: Terminology

RDBMS	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	<code>_id</code>

- ❖ each JSON document:
 - belongs to a collection
 - has a field `_id`
 - unique within the collection
- ❖ each collection:
 - belongs to a “database”





Documents



- ❖ Use **JSON** for API communication
- ❖ Internally: **BSON**
 - **Binary** representation of JSON
 - For storage and inter-server communication
- ❖ Document has a **maximum size**: 16MB (in BSON)
 - Not to use too much RAM
 - GridFS tool can divide larger files into fragments



Document Fields

- ❖ Every **document** must have field **_id**
 - Used as a **primary key**
 - **Unique** within the collection
 - **Immutable**
 - Any **type** other than an array
 - Can be **generated** automatically
- ❖ Restrictions on **field names**:
 - The field names **cannot** start with the **\$** character
 - Reserved for operators
 - The field names **cannot** contain the **.** character
 - Reserved for accessing sub-fields



Database Schema



- ❖ Documents have **flexible schema**
 - Collections do **not enforce** specific data structure
 - In practice, documents in a collection are similar
- ❖ Key **decision** of data modeling:
 - References vs. embedded documents
 - In other words: Where to draw lines between **aggregates**
 - Structure of data
 - Relationships between data



Schema: Embedded Docs

- ❖ Related data in a **single document** structure
 - Documents can have **subdocuments** (in a field or array)

```
{  
  _id: <ObjectId1>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```

Embedded sub-document

Embedded sub-document



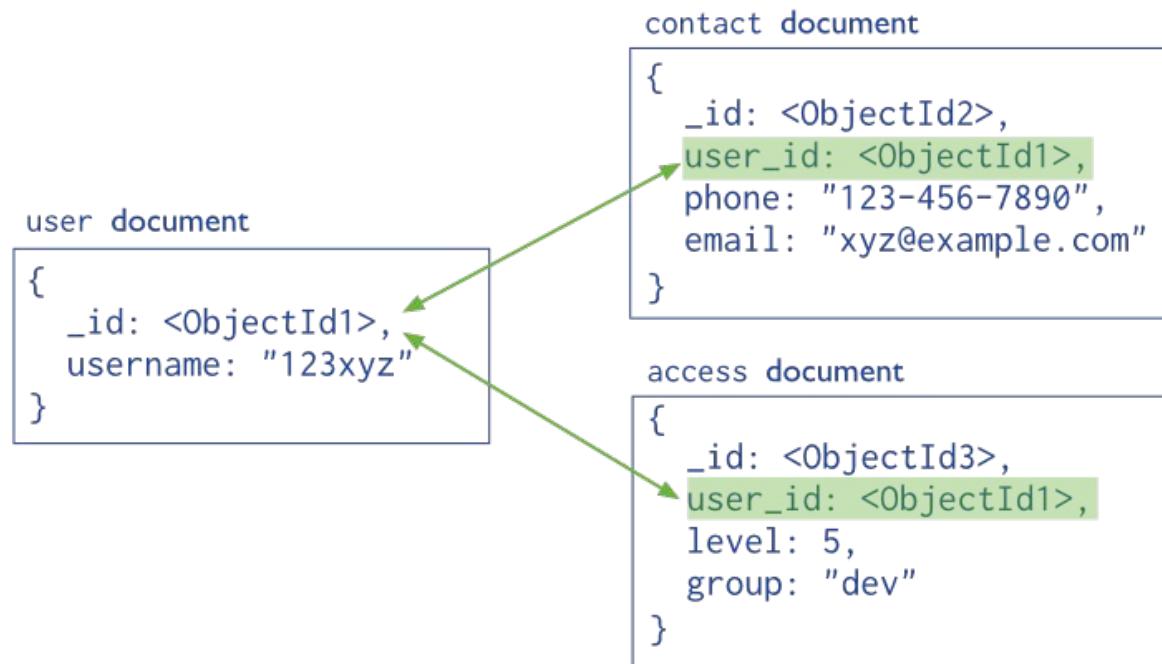
Schema: Embedded Docs (2)

- ❖ Denormalized schema
- ❖ Main advantage:
Manipulate related data in a single operation
- ❖ Use this schema when:
 - One-to-one relationships: one doc “contains” the other
 - One-to-many: if children docs have one parent document
- ❖ Disadvantages:
 - Documents may grow significantly during the time
 - Impacts both read/write performance
 - Document must be relocated on disk if its size exceeds allocated space
 - May lead to data fragmentation on the disk



Schema: References

- ❖ Links/**references** from one document to another
- ❖ **Normalization** of the schema





Schema: References (2)

- ❖ More **flexibility** than embedding
- ❖ **Use references:**
 - When **embedding** would result in **duplication** of data
 - and only insignificant boost of read performance
 - To represent more **complex** many-to-many **relationships**
 - To model large hierarchical data sets
- ❖ Disadvantages:
 - Can require **more roundtrips** to the server
 - Documents are accessed one by one



Querying: Basics

- ❖ Mongo query language
- ❖ A MongoDB **query**:
 - Targets a specific **collection** of documents
 - Specifies **criteria** that identify the returned documents
 - May include a **projection** to **specify** returned **fields**
 - May impose limits, sort, orders, ...
- ❖ Basic query - all documents in the collection:

```
db.users.find()      -- Like SELECT *
```

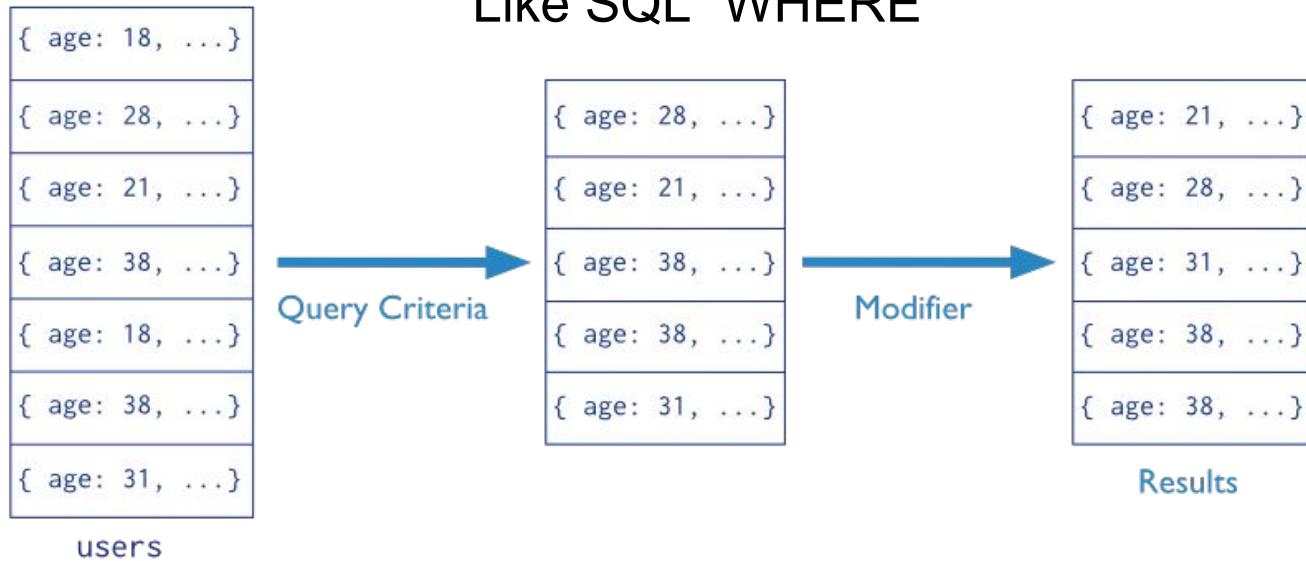
```
db.users.find( {} )
```



Querying: Example

Collection
db.users.find({ age: { \$gt: 18 } }).sort({age: 1})

Like SQL "WHERE"





Querying: Selection

```
db.inventory.find({ type: "snacks" } )
```

- ❖ All documents from collection **inventory** where the **type** field has the value **snacks**

```
db.inventory.find(  
  { type: { $in: [ 'food', 'snacks' ] } } )
```

- ❖ All **inventory** docs where the **type** field is either **food** or **snacks**

```
db.inventory.find(  
  { type: 'food', price: { $lt: 9.95 } } )
```

- ❖ All ... where the **type** field is **food** and the **price** is **less than 9.95**



Inserts



```
db.inventory.insert( { _id: 10, type: "misc",  
item: "card", qty: 15 } )
```

- ❖ Inserts a document with three fields into collection **inventory**
 - User-specified **_id** field

```
db.inventory.insert(  
  { type: "book", item: "journal" } )
```

- ❖ The database generates **_id** field

```
$ db.inventory.find()  
{ "_id": ObjectId("58e209ecb3e168f1d3915300"),  
type: "book", item: "journal" }
```



Updates

```
db.inventory.update(  
  { type: "book", item : "journal" },  
  { $set: { qty: 10 } },  
  { upsert: true } )
```

- ❖ Finds all docs matching query
 - { type: "book", item : "journal" }
- ❖ and sets the field { qty: 10 }
- ❖ upsert: true
 - if no document in the **inventory** collection matches
 - creates a new document (generated _id)
 - it contains fields **_id**, **type**, **item**, **qty**



MapReduce

```
collection "accesses":  
{  
    "user_id": <ObjectId>,  
    "login_time": <time_the_user_entered_the_system>,  
    "logout_time": <time_the_user_left_the_system>,  
    "access_type": <type_of_the_access>  
}
```

- ❖ How much time did **each user** spend logged in
 - Counting just accesses of type “regular”

```
db.accesses.mapReduce(  
    function() { emit (this.user_id, this.logout_time - this.login_time); },  
    function(key, values) { return Array.sum( values ); },  
    {  
        query: { access_type: "regular" },  
        out: "access_times"  
    }  
)
```

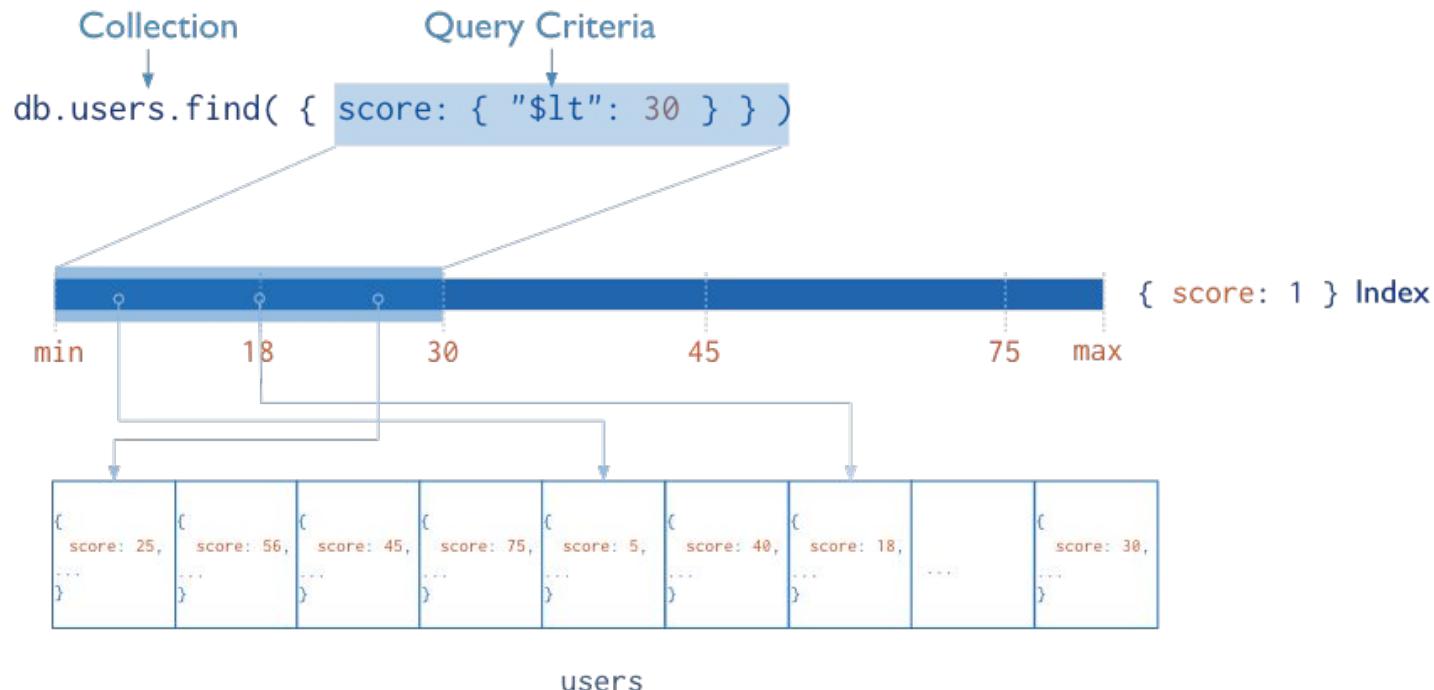


MongoDB Indexes

- ❖ **Indexes** are the key for MongoDB performance
 - **Without** indexes, MongoDB must **scan every** document in a collection to **select** matching documents
- ❖ **Indexes** store some fields in easily accessible form
 - Stores values of a specific field(s) ordered by the value
- ❖ Defined per **collection**
- ❖ Purpose:
 - To **speed up** common queries
 - To optimize **performance** of other specific operations

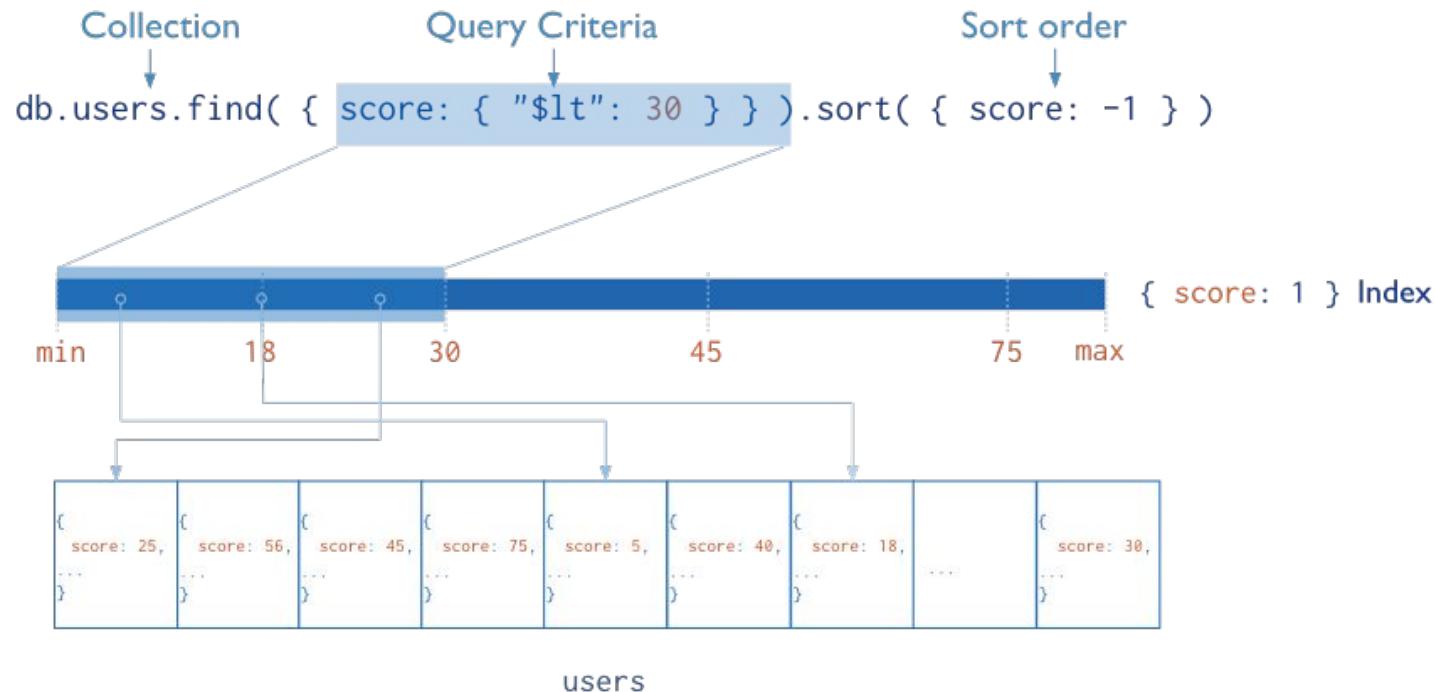


Indexes: Example of Use





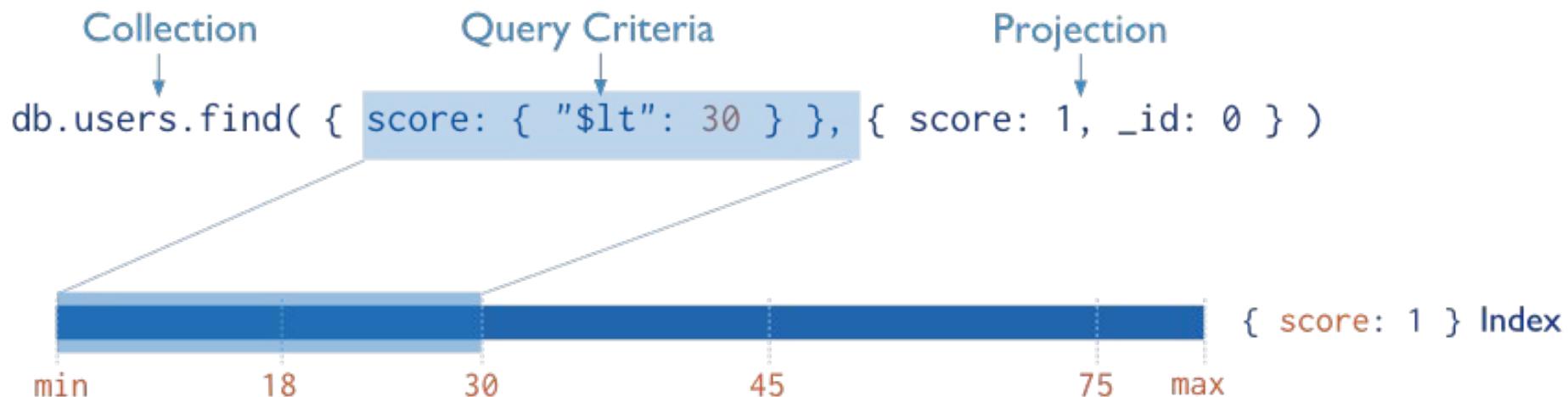
Indexes: Example of Use (2)



- ❖ The **index** can be **traversed** in order to return **sorted results (without sorting)**



Indexes: Example of Use (3)



- ❖ MongoDB does **not** need to inspect data **outside** of the index to fulfill the query

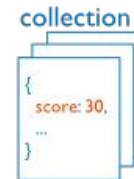


Index Types

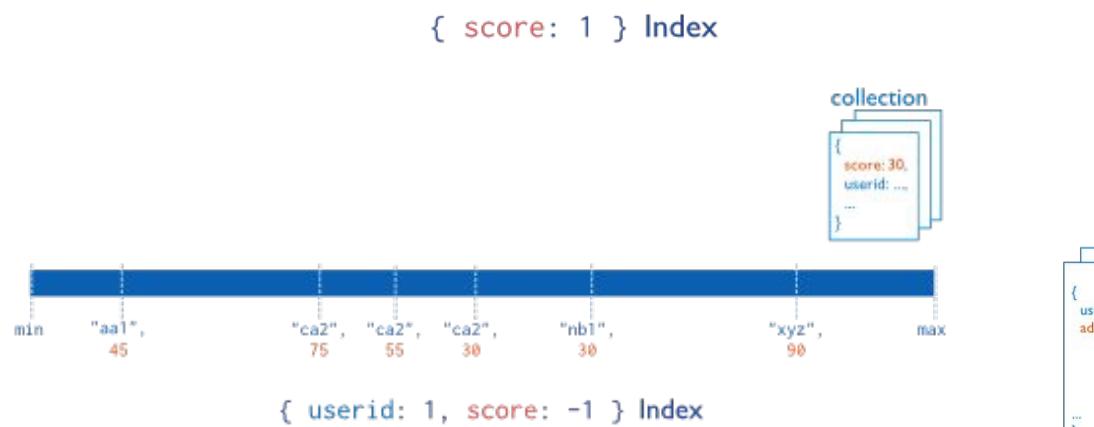
- ❖ **Default: _id**
 - Exists by default
 - If applications do not specify _id, it is created.
 - Unique
- ❖ **Single Field**
 - User-defined indexes on a single field of a document
- ❖ **Compound**
 - User-defined indexes on multiple fields
- ❖ **Multikey index**
 - To index the content stored in arrays
 - Creates separate index entry for each array element



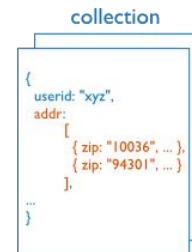
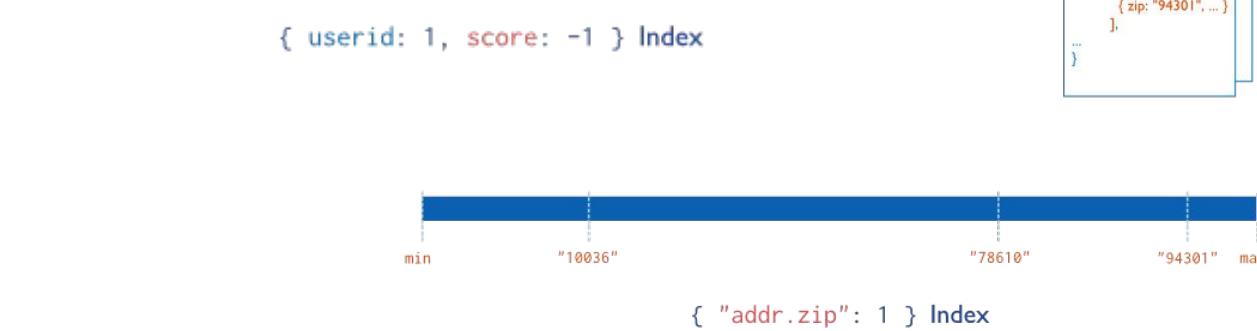
Index Types (2)



- ❖ Index on **score** field (ascending)



- ❖ Compound Index on **userid** (ascending) AND **score** field (descending)



- ❖ Multikey index on the **addr.zip** field



Index Types (3)

- ❖ **Ordered Index**
 - B-Tree (see above)
- ❖ **Hash Indexes**
 - **Fast** $O(1)$ indexes the hash of the value of a field
 - Only **equality** matches
- ❖ **Geospatial Index**
 - 2d indexes = use **planar geometry** when returning results
 - For data representing points on a two-dimensional plane
 - 2sphere indexes = **spherical** (Earth-like) geometry
 - For data representing longitude, latitude
- ❖ **Text Indexes**
 - Searching for **string** content in a collection



MongoDB: Behind the Curtain

- ❖ **BSON** format
- ❖ **Distribution** models
 - Replication
 - Sharding
 - Balancing
- ❖ MapReduce
- ❖ Transactions
- ❖ Journaling



BSON (*Binary JSON*) Format

- ❖ **Binary-encoded serialization** of JSON documents
 - Representation of documents, arrays, JSON simple data types + other types (e.g., date)

```
{"hello": "world"}
```

→ "\x16\x00\x00\x00\x02hello\x00
 \x06\x00\x00\x00world\x00\x00"

```
{"BSON": ["awesome",  
5.05, 1986]}
```

→ "\x31\x00\x00\x00\x04BSON\x00\x26\x00
 \x00\x00\x020\x00\x08\x00\x00
 \x00awesome\x00\x011\x00\x33\x33\x33
 \x33\x33\x33
 \x14\x40\x102\x00\xc2\x07\x00\x00
 \x00\x00"



BSON: Basic Types

- ❖ byte – 1 byte (8-bits)
- ❖ int32 – 4 bytes (32-bit signed integer)
- ❖ int64 – 8 bytes (64-bit signed integer)
- ❖ double – 8 bytes (64-bit IEEE 754 floating point)



BSON Grammar

```
document ::= int32 e_list "\x00"
```

- ❖ **BSON document**
- ❖ **int32 = total number of *bytes* in document**

```
e_list ::= element e_list | ""
```

- ❖ **Sequence of elements**



BSON Grammar (2)

```
element ::= "\x01" e_name double  
| "\x02" e_name string  
| "\x03" e_name document  
| "\x04" e_name document  
| "\x05" e_name binary  
| ...
```

Floating point
UTF-8 string
Embedded document
Array
Binary data
...

e_name ::= cstring

- **Field key**

cstring ::= (byte*) "\x00"

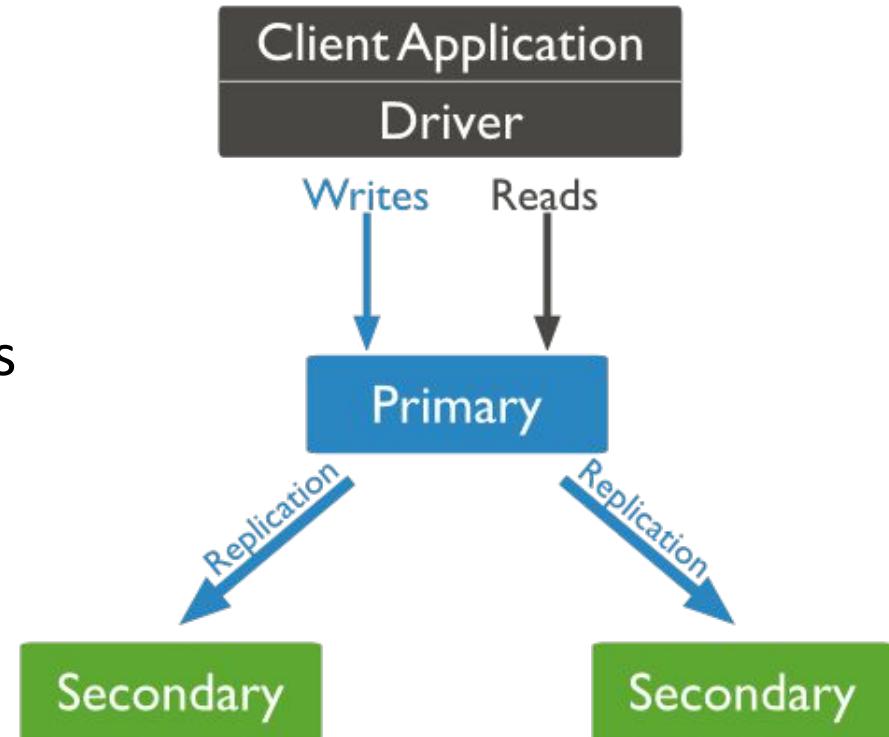
string ::= int32 (byte*) "\x00"

etc....



Data Replication

- ❖ Master/slave replication
- ❖ **Replica set** = group of instances that host the **same data set**
 - **primary** (master) – handles all **write** operations
 - **secondaries** (slaves) – apply operations from the primary so that they have the same data set





Replication: Read & Write

- ❖ **Write operation:**
 1. Write operation is applied on the **primary**
 2. Operation is recorded to primary's **oplog** (operation log)
 3. **Secondaries** replicate the **oplog** + **apply** the operations to their data sets
- ❖ **Read:** All replica set **members** can accept **reads**
 - By **default**, application directs its reads to the primary
 - Guarantees the latest version of a document
 - **Decreases** read **throughput**
 - **Read preference** mode can be **set**
 - See below



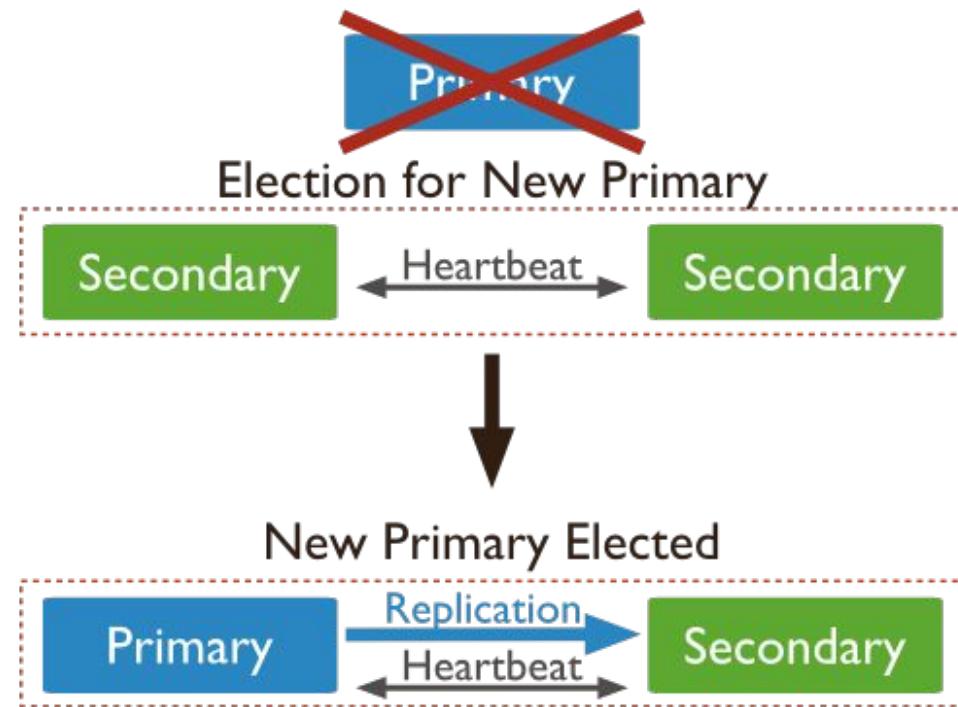
Replication: Read Modes

Read Preference Mode	Description
primary	operations read from the primary of the replica set
primaryPreferred	operations read from the primary , but if unavailable, operations read from secondary members
secondary	operations read from the secondary members
secondaryPreferred	operations read from secondary members, but if none is available, operations read from the primary
nearest	operations read from the nearest member (= shortest ping time) of the replica set



Replica Set Elections

- ❖ If the **primary** becomes **unavailable**, an election determines a **new primary**
 - Elections need some time
 - No primary => no writes





Replica Set: CAP

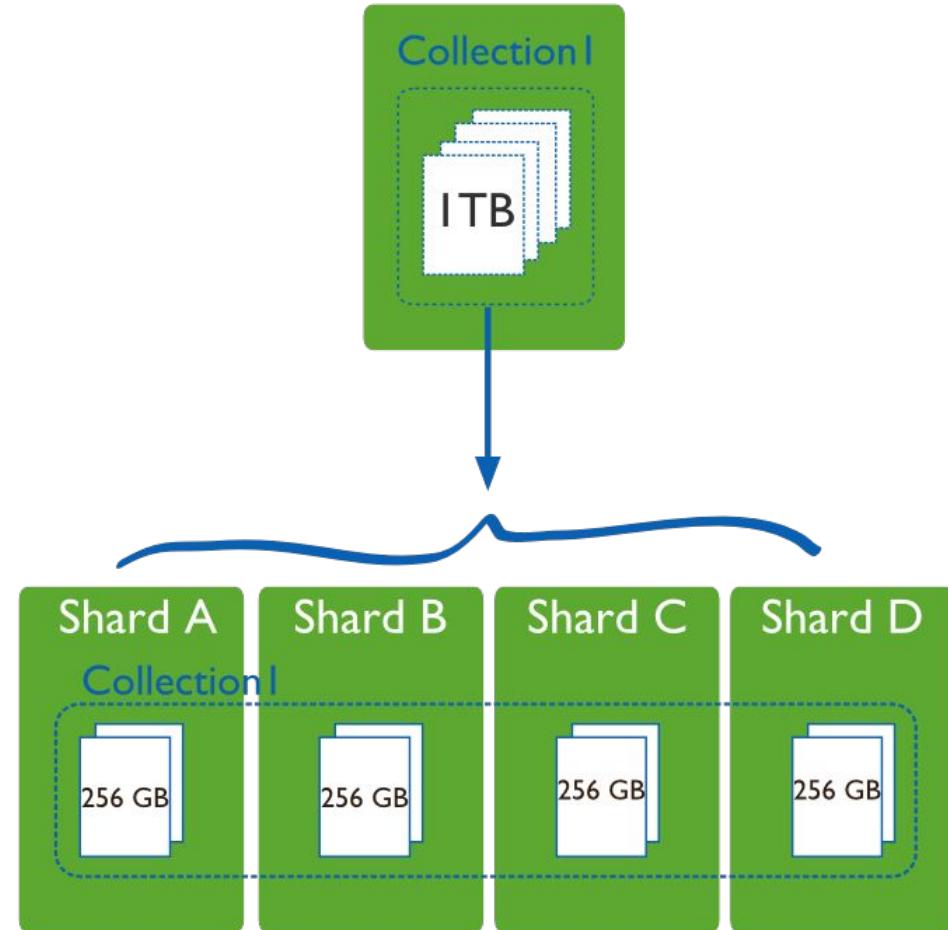
- ❖ Let us have **three** nodes in the **replica set**
 - Let's say that the **master** is **disconnected** from the other two
 - The distributed system is **partitioned**
 - The **master** finds out, that it is **alone**
 - Specifically, that can communicate with **less than half** of the nodes
 - And it steps down from being master (handles just reads)
 - The other two slaves “think” that the **master failed**
 - Because they form a partition with **more than half** of the nodes
 - And elect a new master
- ❖ In case of just **two nodes** in RS
 - **Both partitions will become read-only**
 - Similar case can occur with any **even number of nodes** in RS
 - Therefore, we can always **add** an **arbiter** node to an even RS



Sharding



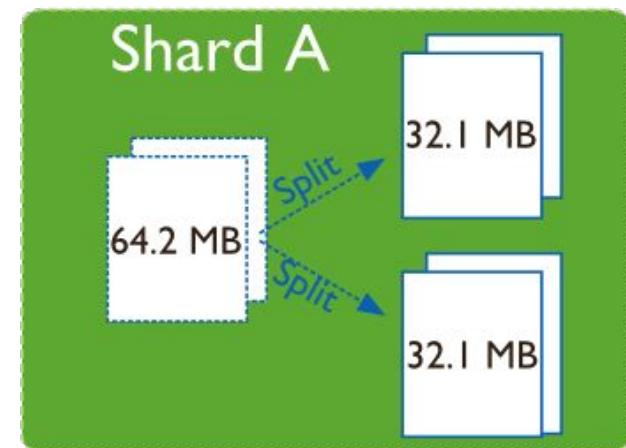
- ❖ MongoDB enables **collection partitioning** (sharding)





Collection Partitioning

- ❖ Mongo partitions collection's data by the **shard key**
 - Indexed **field(s)** that exist in **each document** in the collection
 - Immutable
 - **Divided** into chunks, distributed across shards
 - Range-based partitioning
 - Hash-based partitioning
 - When a chunk grows **beyond** the size **limit**, it is **split**
 - Metadata change, **no** data **migration**
- ❖ Data **balancing**:
 - Background chunk migration



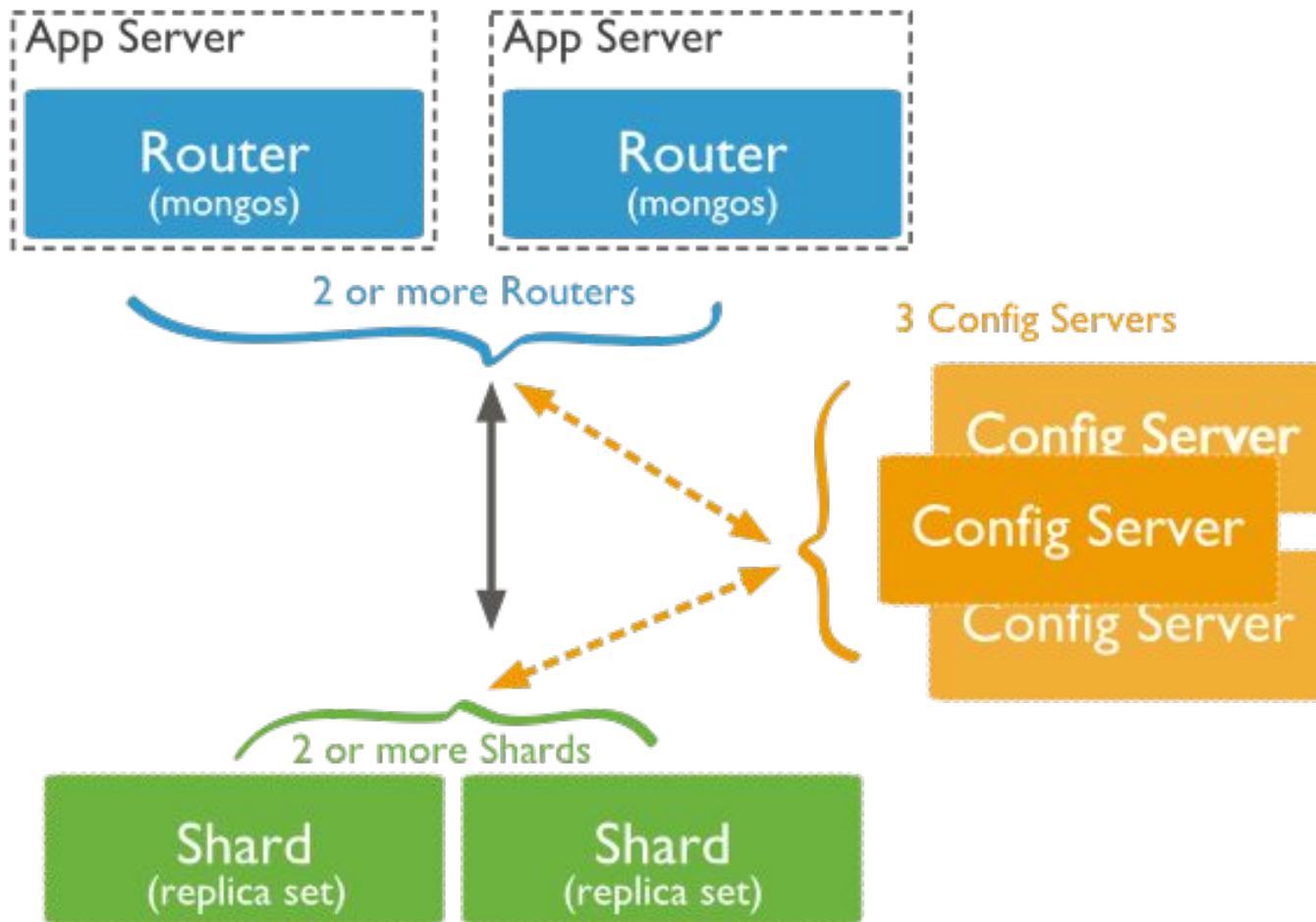


Sharding: Components

- ❖ MongoDB runs in **cluster** of different node types:
- ❖ **Shards** – store the data
 - Each **shard** is a replica set
 - Can be **a single node**
- ❖ **Query routers** – interface with client applications
 - **Direct** operations **to** the **relevant** shard(s)
 - + **return** the result to the client
 - More than one => to divide the client request load
- ❖ **Config servers** – store the cluster's metadata
 - **Mapping** of the cluster's data set to the shards
 - Recommended number: 3



Sharding: Diagram





Journaling

- ❖ **Write** operations are applied **in memory and into a journal** before done in the data files (on disk)
 - To **restore** consistent state after a **hard shutdown**
 - Can be switched on/off
- ❖ **Journal directory** – holds journal files
- ❖ **Journal file** = write-ahead **redo logs**
 - Append only file
 - **Deleted** when all the writes are **durable**
 - When size > 1GB of data, MongoDB creates a new file
 - The size can be modified
- ❖ **Clean shutdown** removes all journal files



Transactions

- ❖ Write ops: **atomic** at the level of **single document**
 - Including nested documents
 - Sufficient for many cases, but not all
 - When a write operation modifies **multiple** documents, **other** operations may **interleave**
- ❖ **Transactions:**
 - **Isolation** of a write operation that affects multiple docs

```
db.foo.update( { field1 : 1 , $isolated : 1 } , { $inc : { field2 : 1 } } , { multi: true } )
```
 - **Two-phase commit**
 - Multi-document updates



NoSQL *Columnar Databases*



Problem Set #5 is due tonight
Problem Set #6 will be online tonight, or you all get 100.



Agenda

- ❖ Data Model
 - Column **families**, super columns, two points of view
- ❖ Column-family Stores
 - Google BigTable, Cassandra, HBase
- ❖ Cassandra as an Example
 - Cassandra data model 1.0 vs. 2.0
 - Cassandra Query Language (CQL)
 - Data **partitioning**, replication
 - Local Data **Persistence**
 - **Query** processing, Indexes, Lightweight Transactions



Column-family Stores: Basics

- ❖ AKA: wide-column, columnar
 - not to **confuse** with column-oriented RDBMS
- ❖ Data model: **rows** that have **many columns** associated with a **row key**
- ❖ **Column families** are groups of related data (columns) that are often **accessed together**
 - e.g., for a **customer** we typically access all **profile** information at the same time, but not customer's **orders**



Data Model

- ❖ *Columns within Rows = the basic data item*

- a **3-tuple** consisting of

- column **name**
 - **value**
 - **timestamp**

- Can be **modeled** as follows

```
{ name: "firstName",  
  value: "Martin",  
  timestamp: 12345667890 }
```

column_name
value
timestamp

- ❖ In the following, we will **ignore** the **timestamp**



Data Model

- ❖ **Row:** a collection of columns with a common **row key**
 - Columns can be **added to** any **row** at any time
 - without having to add it to other rows

```
// row
```

```
"martin-fowler" : { ←———— Row key
```

Column
keys {
 firstName: "Martin",
 lastName: "Fowler",
 location: "Boston"
}

Row key1	Column Key1	Column Key2	Column Key3	...
	Column Value1	Column Value2	Column Value3	
				:



Data Model: Column Family

- ❖ CF = Set of columns containing “related” data

user_id (row key)	column key	column key	...
	column value	column value	...
1	login	first_name	...
	gonzo	Sam	...
4	login	age	...
	david	35	...
5	first_name	last_name	...
	Kathy	Wright	...
...			



Data Model: Column Family (2)

- ❖ Column family - example as JSON

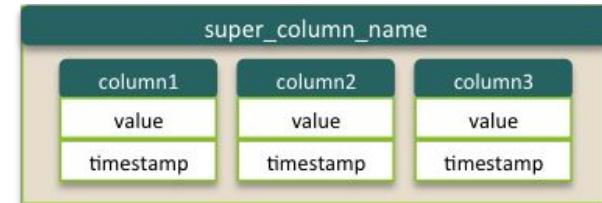
```
{ // row (columns from a CF)      // row (cols from the same CF)
  "pramod-sadalage" : {           "martin-fowler" : {
    firstName: "Pramod",          firstName: "Martin",
    lastName: "Sadalage",         lastName: "Fowler",
    lastVisit: "2012/12/12"       location: "Boston",
                                activ: "true"  }
  }
}
```



Data Model: Super Column Family

❖ Super column

- A **column** whose value is composed of a **map of columns**
- Used in some column-family stores (Cassandra 1.0)



❖ Super column family

- A column family consisting of super columns

Row key1	Super Column key1			Super Column key2			...
	Subcolumn Key1	Subcolumn Key2	...	Subcolumn Key3	Subcolumn Key4	...	
	Column Value1	Column Value2	...	Column Value3	Column Value4	...	



Super Column Family: Example

user_id (row key)	super column key			super column key			...
	subcolumn key	subcolumn key	...	subcolumn key	subcolumn key	...	
	subcolumn value	subcolumn value	...	subcolumn value	subcolumn value	...	
1	home_address			work_address			
	city	street	...	city	street	...	
	Raleigh	Hillsborough St	...	Chapel Hill	Raleigh St	...	
4	home_address			temporary_address			
	city	street	...	city	street		
	Durham	Chapel Hill St	...	Chapel Hill	Raleigh St		
...							



Super Column Family in JSON

```
{ // row
  "Cathy": {
    "username": { "firstname": "Cathy", "lastname": "Qi" },
    "address": { "city": "New York", "zip": "10001" }
  }
  // row
  "Terry": {
    "username": { "firstname": "Terry", "lastname": "Martin" },
    "account": { "bank": "Citi", "account": 12346789 },
    "background": { "birthdate": "1990-03-04" }
  }
}
```



Column Family Stores: Features



- ❖ Data **model**: Column families
- ❖ System architecture
 - Data partitioning
- ❖ Local **persistence**
 - update log, memory, disk...
- ❖ Data **replication**
 - balancing of the data
- ❖ Query processing
 - query language
- ❖ Indexes



Representatives





BigTable



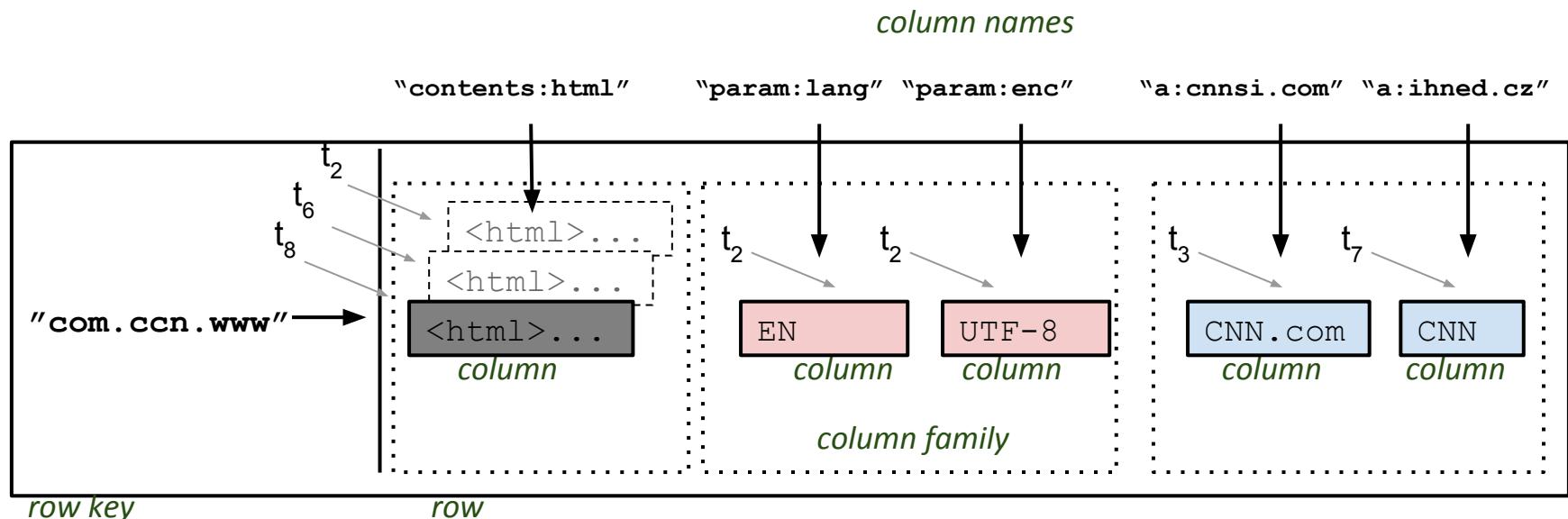
- ❖ Google's **paper**:
 - Chang, F. et al. (2008). Bigtable: A Distributed Storage System for Structured Data. ACM TOCS, 26(2), pp 1–26.
- ❖ Proprietary, not distributed outside Google
 - used in Google Cloud Platform
- ❖ Data **model**: column families as defined above
 - “A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map.”

(row:string, column:string, time:int64) → string



BigTable: Example

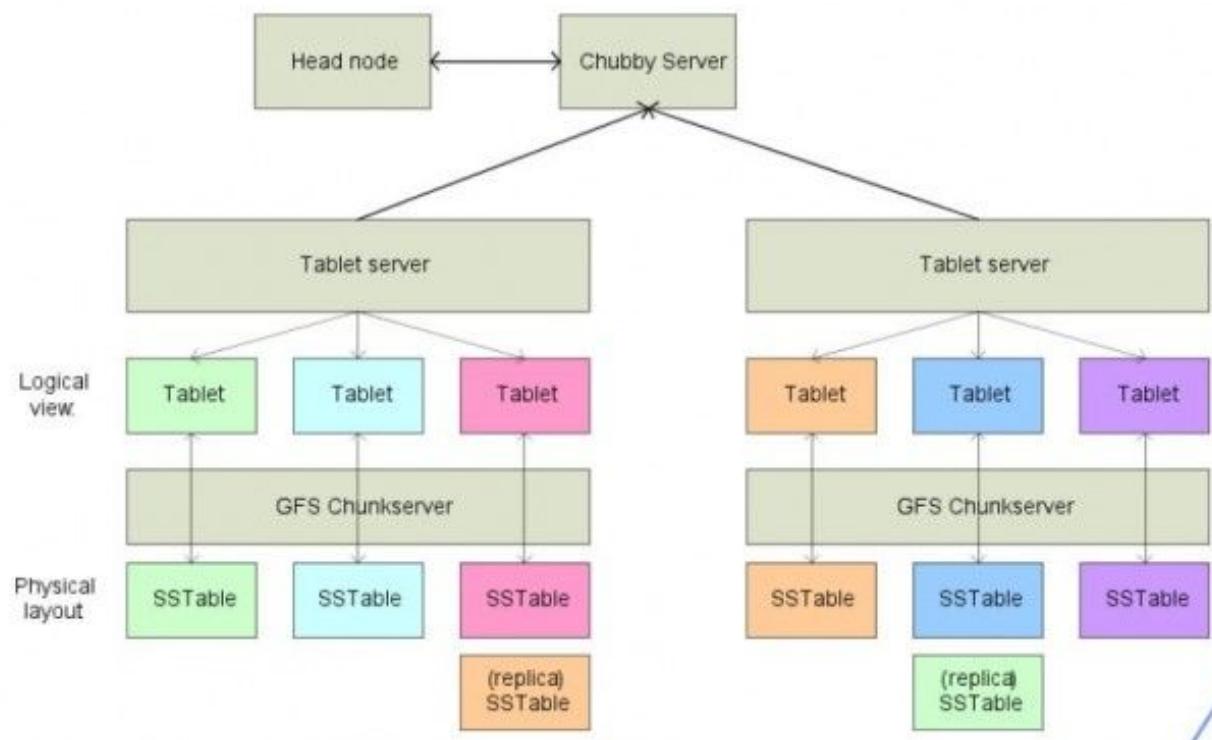
- “BigTable = sparse, distributed, persistent, multi-dimensional sorted map indexed by (*row_key*, *column_key*, *timestamp*)”





BigTable: Architecture

Bigtable Architecture





Cassandra



- ❖ Developed at **Facebook**
 - now, Apache Software License 2.0
- ❖ Initial release: 2008
- ❖ Written in: **Java**
- ❖ OS: cross-platform
- ❖ Operations:
 - **CQL** (Cassandra Query Language)
 - **MapReduce** support (can cooperate with Hadoop)



Cassandra: Data Model



- ❖ **Column** families, **super** column families
 - Can define metadata about columns
 - Now denoted as: Thrift API
- ❖ **Static** – similar to a relational database table
 - **Rows** have the **same** limited set of **columns**
 - However, rows are **not required** to have **all** columns
- ❖ **Dynamic** – takes advantage of Cassandra's ability to use **arbitrary new column names**



Cassandra: Data Model

row key	columns...									
Alex Smith	2010	2011	2012	2013	2014	2015	2016	2017	2018	
	49ers	49ers	chiefs	redskins						
Blaine Gabbert	2011	2012	2015	2016						
	jaguars	jaguars	49ers	49ers						
Colin Kaepernick	2012	2013	2014	2015	2016					
	49ers	49ers	49ers	49ers	49ers					

row key	columns...							
24911	first	last	college	dob	height	weight	2016	
	Alex	Smith	Utah	1984-05-07	6-4	212	chiefs	
27073	first	last	college	dob	high school			
	Blaine	Gabbert	Missouri	1989-10-15	Parkway West			
27154	first	last	college	height	weight			
	Colin	Kaepernick	Nevada	6-4	225			



Key Spaces

- ❖ Databases are called "KEYSPACES" in Cassandra
- ❖ They are created as follows:

```
CREATE KEYSPACE db
    WITH replication = {
        'class' : 'SimpleStrategy',
        'replication_factor': '2' }
    AND durable_writes = 'true';
```

```
DESCRIBE KEYSPACES;
```



Cassandra Tables



KEYSPACES *can* have TABLES defined as follows

```
CREATE TABLE Player (
    pid int PRIMARY KEY,
    first TEXT,
    last TEXT,
    college TEXT);
```

```
DESCRIBE TABLES;
```



Working with Tables

```
CREATE TABLE users (
    user_id int PRIMARY KEY,
    login text,
    name text,
    email text );

INSERT INTO users (user_id, login, name)
VALUES (3, 'cathyqi', 'Cathy Qi');

SELECT * FROM users;
user_id | email | login | name
-----+-----+-----+-----
      3 | null | cathyqi | Cathy Qi
```



Cassandra: Column Families

- ❖ An alternative to tables are *COLUMN FAMILIES*
- ❖ Requires a Name and Comparators
 - A **key** *must* be specified
 - Data **types** for columns *can* be specified
 - Options **can** be specified

```
CREATE COLUMNFAMILY Fish (key blob PRIMARY KEY);
```

```
CREATE COLUMNFAMILY FastFoodPlaces (name text PRIMARY KEY)  
    WITH comparator=timestamp AND default_validation=int;
```

```
CREATE COLUMNFAMILY MonkeyTypes (  
    key uuid PRIMARY KEY,  
    species text,  
    alias text,  
    population varint  
) WITH comment='Important biological records'  
    AND read_repair_chance = 1.0;
```



Cassandra: Column Families (2)



- ❖ **Comparator** = data type for a **column name**
- ❖ **Validator** = data type of a **column value**
 - or content of a **row key**
- ❖ Data types do **not need** to be defined
 - Default: BytesType, i.e. arbitrary hexadecimal bytes
- ❖ Basic operations: GET, SET, DEL



Cassandra: Data Manipulation

```
create column family users
  with key_validation_class = Int32Type
  and comparator = UTF8Type
  and default_validation_class = UTF8Type;
```

```
// set column values in row with key 7
set users[7]['login'] = utf8('cathyqi');
set users[7]['name'] = utf8('Cathy Qi');
set users[7]['email'] = utf8('qi@best.com');

set users[13]['login'] = utf8('fantom');
set users[13]['name'] = utf8('Un Known');
```



Cassandra: Data Manipulation (2)



```
get users[7]['login'];
=> (name=login, value=cathyqi, timestamp=1429268223462000)

get users[13];
=> (name=login, value=fantom, timestamp=1429268224554000)
=> (name=name, value=Un Known, timestamp=1429268224555000)

list users;
RowKey: 7
=> (name=email, value=qi@best.com, timestamp=14292682...)
=> (name=login, value=cathyqi, timestamp=1429268223462000)
=> (name=name, value=Cathy Qi, timestamp=1429268223471000)
-----
RowKey: 13
=> (name=login, value=fantom, timestamp=1429268224554000)
=> (name=name, value=Un Known, timestamp=1429268225231000)
```



Cassandra: Sparse Tables

- ❖ CQL: Cassandra Query Language
 - **SQL-like** commands
 - CREATE, ALTER, UPDATE, DROP, DELETE, TRUNCATE, INSERT, ...
 - **Simpler** than SQL
- ❖ Since CQL 3 (Cassandra 1.2)
 - Column -> **cell**
 - Column **family** -> **table**
- ❖ **Dynamic** columns (wide rows) still **supported**
 - CQL supports everything that was possible before
 - “**Old**” approach (Thrift API) **can** be used as well



Tables: Dynamic Columns



- ❖ **Values** can use “collection” types:
 - **set** – unordered unique values
 - **list** – ordered list of elements
 - **map** – name + value pairs
 - a way to **realize super-columns**
- ❖ **Realization** of the original idea of **free columns**
 - Internally, all **values** in collections as individual **columns**
 - Cassandra can well **handle** “unlimited” number of columns



Tables: Dynamic Columns (2)

```
CREATE TABLE users (
    login text PRIMARY KEY,
    name text,
    emails set<text>, // column of type "set"
    profile map<text, text> // column of type "map"
)

INSERT INTO users (login, name, emails, profile)
VALUES ( 'honza', 'Jan Novák', { 'honza@novak.cz' },
         { 'colorschema': 'green', 'design': 'simple' }
) ;

UPDATE users
SET emails = emails + { 'jn@firma.cz' }
WHERE login = 'honza';
```



Dynamic Columns: Another Way

❖ Compound primary key

```
CREATE TABLE mytable (
    row_id int, column_name text, column_value text,
    PRIMARY KEY (row_id, column_name)
);
```

```
INSERT INTO mytable (row_id, column_name, column_value)
VALUES ( 3, 'login', 'honza' );
```

```
INSERT INTO mytable (row_id, column_name, column_value)
VALUES ( 3, 'name', 'Jan Novák');
```

```
INSERT INTO mytable (row_id, column_name, column_value)
VALUES ( 3, 'email', 'honza@novak.cz' );
```



Data Sharding in Columnar Systems

System	Terminology
BigTable	tablets
HBase	regions
Cassandra	partitions

user_id (row key)	login	name
1	cathyqi	Cathy Qi
4	davidm	David Man
...		
1000	iamboo	Nota James
1001	violet	Ziwei Chen
1003	ernie	Bernard ...
...		
2000	peach	Joseph Ash
...		



Data Sharding in Cassandra

- ❖ Entries in each table are **split** by **partition key**
 - Which is a selected **column** (or a set of columns)
 - Specifically, the **first column** (or columns) from the **primary key** is the **partition key** of the table

```
CREATE TABLE tab ( a int, b text, c text, d text,  
    PRIMARY KEY ( a, b, c)  
);
```

```
CREATE TABLE tab ( a int, b text, c text, d text,  
    PRIMARY KEY ( (a, b), c)  
);
```



Data Sharding in Cassandra (2)



- ❖ All entries with the same **partition key**
 - Will be stored on the **same physical node**
 - => **efficient** processing of **queries** on one partition key

```
CREATE TABLE mytable (
    row_id int, column_name text, column_value text,
    PRIMARY KEY (row_id, column_name) );
```

- ❖ The **rest** of the columns in the primary key
Are so called **clustering columns**
 - Rows are **locally sorted** by values in the **clustering columns**
 - the order for **physical storing** rows



Data Replication



- ❖ Cassandra adopts peer-to-peer **replication**
 - The **same principles** like in key-value stores & document DB
 - Read/Write **quora** to balance between **availability** and **consistency** guarantees
- ❖ Google BigTable
 - Physical data **distribution** & replication is done by the underlying **distributed file system**
 - GFS



Cassandra Query Language (CQL)



- ❖ The **syntax** of CQL is **similar** to SQL
 - But search just in **one table** (no joins)

```
SELECT <selectExpr>
FROM [<keyspace>.]<table>
[WHERE <clause>]
[ORDER BY <clustering_colname> [DESC] ]
[LIMIT m];
```

```
SELECT column_name, column_value
FROM mytable
WHERE row_id=3
ORDER BY column_value;
```



CQL: Limitations on “Where” Part

- ❖ The search condition can be:
 - on columns in the **partition key**
 - And only using **operators == and IN**
 - ... WHERE row_id IN (3, 4, 5)
 - Therefore, the query hits only **one or several physical nodes** (not all)
 - on columns from the **clustering key**
 - Especially, if there is also condition on the **partitioning key**
 - ... WHERE row_id=3 AND column_name='login'
 - If it is not, the system must **filter all entries**

```
CREATE TABLE mytable (
    row_id int,
    column_name text,
    column_value text,
    PRIMARY KEY
        (row_id, column_name)
);
```

```
SELECT * FROM mytable
WHERE column_name IN ('login', 'name') ALLOW FILTERING;
```



CQL: Limitations on “Where” Part (1)

- ❖ Other **columns** can be queried
 - If there is an **index** built on the column
- ❖ **Indexes** can be built also on **collection** columns (set, list, map)
 - And then **queried** by CONTAINS like this

```
SELECT login FROM users  
    WHERE emails CONTAINS 'jn@firma.cz';
```

```
SELECT * FROM users  
    WHERE profile CONTAINS KEY 'colorschema';
```



Indexes



- ❖ **Secondary indexes on any column**
 - B⁺-Tree indexes
 - **User-defined implementation of indexes**

```
CREATE INDEX ON users (emails);
```



Transactions

- ❖ Cassandra 2.x supports “**lightweight** transactions”
 - **compare and set** operations
 - using **Paxos** consensus protocol
 - nodes **agree** on proposed data additions/modifications
 - **faster** than Two-phase commit protocol (P2C)

```
INSERT INTO users (login, name, emails)
VALUES ('cathyqi', 'Cathy Qi', { 'qi@best.com' })
IF NOT EXISTS;
```

```
UPDATE mytable SET column_value = 'qi@best.org'
WHERE row_id = 3 AND column_name = 'email'
IF column_value = 'qi@best.com';
```



Summary



❖ Column-family stores

- are worth only for **large data** and large query **throughput**
- two ways to see the **data model**:
 - large sparse **tables** or multidimensional (nested) **maps**
- data distribution is via row key
 - analogue of **document ID** or **key** in **document** or **key-value stores**
- efficient disk + memory local data storage

❖ Cassandra

- CQL: structured after SQL, easy transition from RDBMS



NoSQL

Graph Databases



Problem Set #4 is graded
Problem Set #6 is done, you will all get 100!



Agenda



- ❖ Graph Databases: Mission, Data, Example
- ❖ A Bit of Graph Theory
 - Graph Representations
 - Algorithms: Improving Data Locality (efficient storage)
 - Graph Partitioning and Traversal Algorithms
- ❖ Graph Databases
 - Transactional databases
 - Non-transactional databases
- ❖ Neo4j
 - Basics, Native Java API, Cypher, Behind the Scene



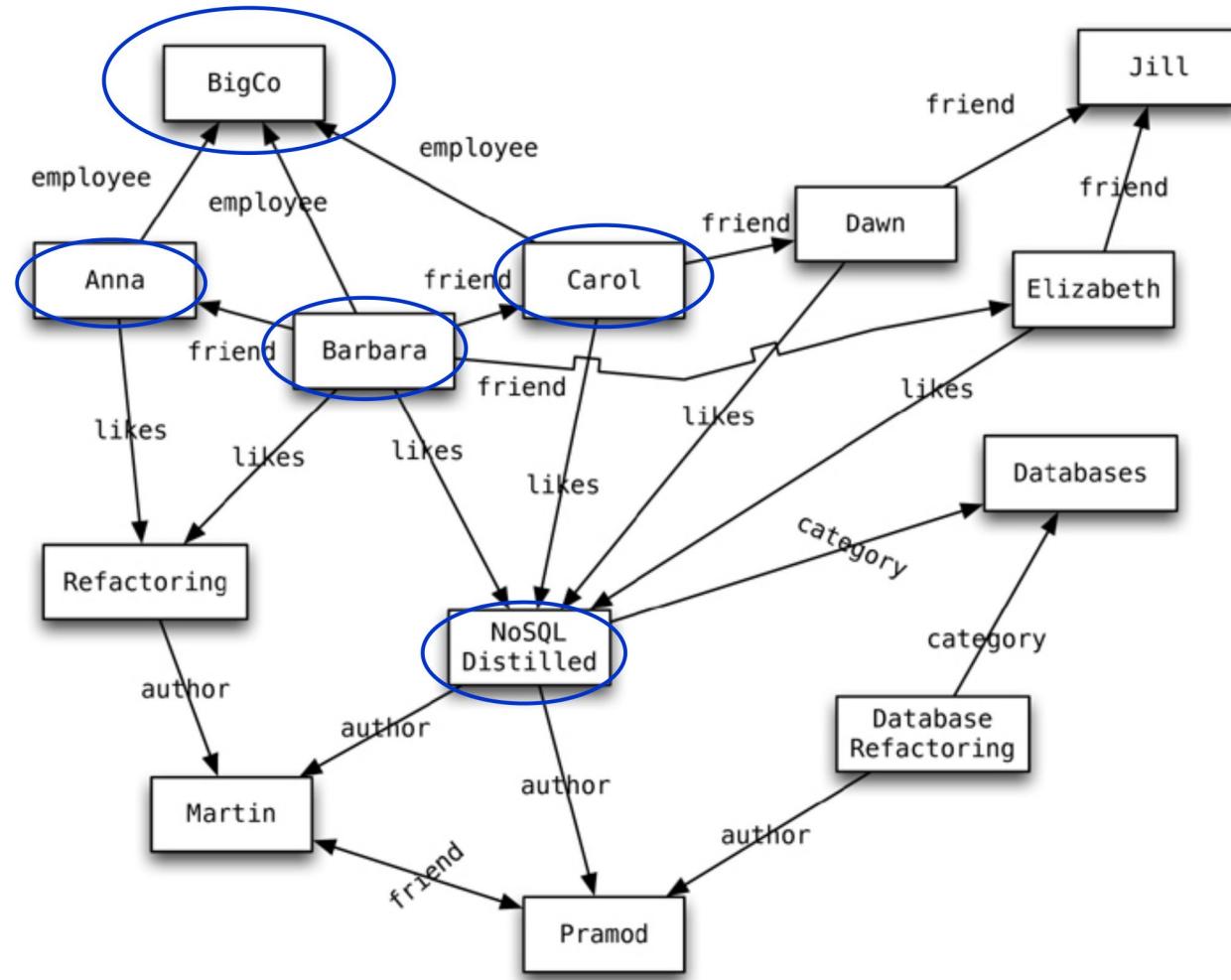
Graph Databases: Concept



- ❖ To store **entities** and **relationships** between them
 - **Nodes** are instances of objects
 - Nodes have **properties**, e.g., name
 - **Edges** connect nodes and are **directed**
 - Edges have **types** (e.g., likes, friend, ...)
- ❖ Nodes are organized by **relationships**
 - Allow to **find** interesting **patterns**
 - **example:** Get all nodes that are “employee” of “Big Company” and that “likes” “NoSQL Distilled”

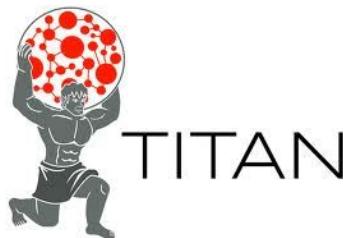


Graph Databases: Example





Graph Databases: Representatives





Graph Database Basics

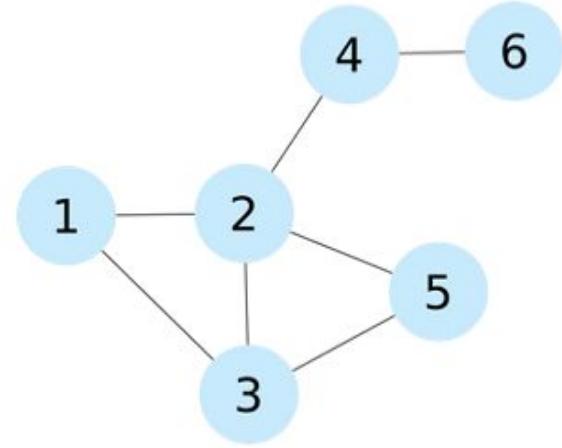
- ❖ Data: a **set** of entities and their **relationships**
 - => we need to **efficiently represent graphs**
- ❖ Basic **operations**:
 - finding the **neighbours** of a node,
 - **checking** if two nodes are connected by an edge,
 - **updating** the graph structure, ...
 - => we need **efficient graph operations**
- ❖ Graph $G = (V, E)$ is usually **as
 - set of **nodes** (vertices) V , $|V| = n$
 - set of **edges** E , $|E| = m$**
- ❖ Which **to use?**



Data Structure: Adjacency Matrix

- ❖ Two-dimensional **array A** of $n \times n$ Boolean values
 - **Indexes** of the array = **node identifiers** of the graph
 - Boolean value A_{ij} indicates whether nodes i, j are **connected**

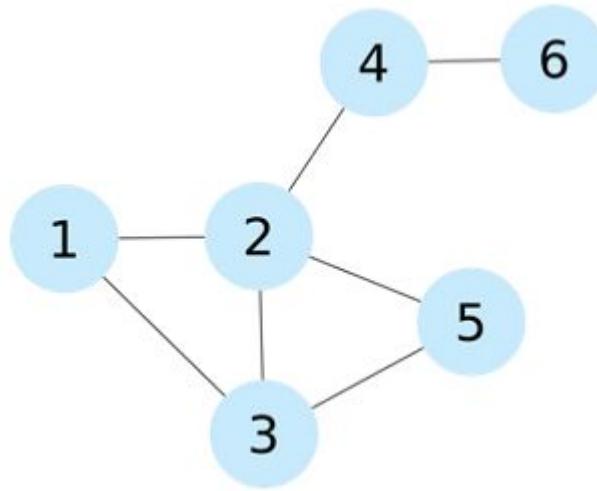
- ❖ **Variants:**
 - (Un)directed graphs
 - Weighted graphs...



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	1	0
4	0	1	0	0	0	1
5	0	1	1	0	0	0
6	0	0	0	1	0	0



Adjacency Matrix: Properties



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	1	0
4	0	1	0	0	0	1
5	0	1	1	0	0	0
6	0	0	0	1	0	0

❖ Pros:

- Adding/removing **edges**
- **Checking** if 2 nodes are connected

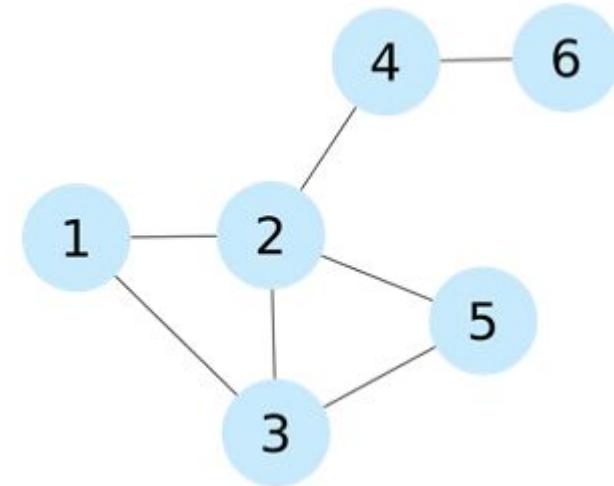
❖ Cons:

- Quadratic **space**: $O(n^2)$
- **Sparse** graphs (mostly 0s) are common
- **Adding nodes** is expensive
- Retrieval the **neighbouring nodes** takes linear time: $O(n)$



Data Structure: Adjacency List

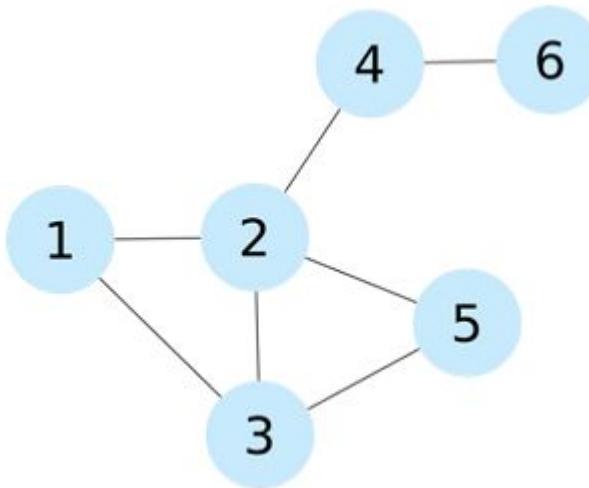
- ❖ A **dictionary** or list of **lists**,
describing the **neighbours**
of the **key** or indexed node
 - Vector of n pointers to adjacency lists
- ❖ **Undirected graph:**
 - An edge connects nodes i and j
 - => the adjacency list of i contains
node j and **vice versa**
- ❖ Often **compressed**
 - Exploiting **regularities** in graphs



Neighbors[1] =	[2, 3]
Neighbors[2] =	[1, 3, 5]
Neighbors[3] =	[1, 2, 5]
Neighbors[4] =	[2, 6]
Neighbors[5] =	[2, 3]
Neighbors[6] =	[4]



Adjacency List: Properties



❖ Pros:

- Getting the neighbours of a node
- Cheap **addition of nodes**
- More **compact** representation of **sparse** graphs

❖ Cons:

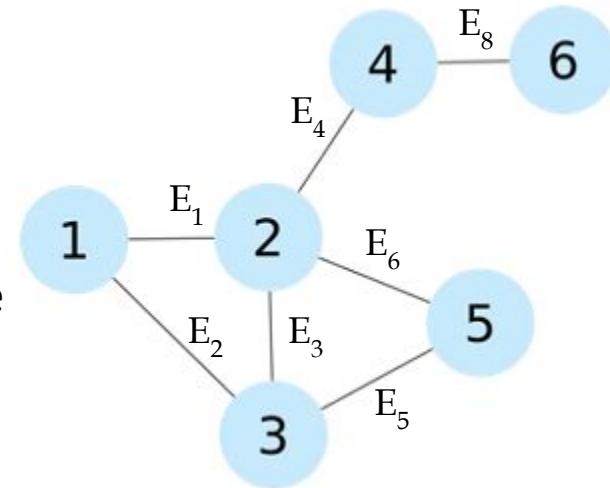
- **Checking if an edge exists between two nodes**
 - **Optimization:** sorted lists => logarithmic scan, but also logarithmic insertion

```
Neighbors[1] = [2, 3]
Neighbors[2] = [1, 3, 5]
Neighbors[3] = [1, 2, 5]
Neighbors[4] = [2, 6]
Neighbors[5] = [2, 3]
Neighbors[6] = [4]
```



Data Structure: Incidence Matrix

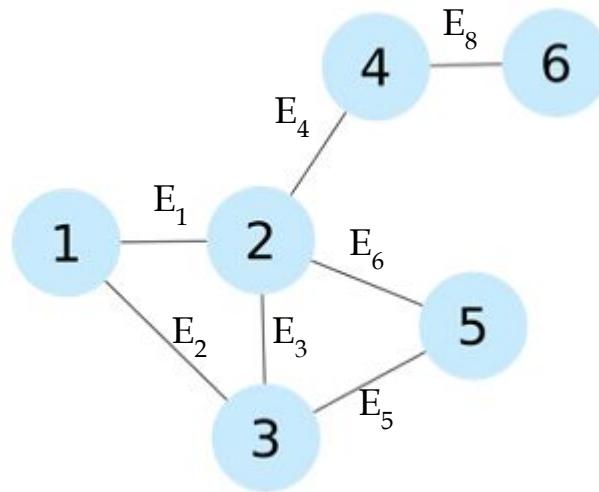
- ❖ Two-dimensional Boolean **matrix** of ***n*** rows and ***m*** columns
 - A **column** represents an **edge**
 - Nodes that are connected by a certain edge
 - A **row** represents a **node**
 - All edges that are connected to the node



	E ₁	E ₂	E ₃	E ₄	E ₅	E ₇	E ₈
1	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0
3	0	1	1	0	0	1	0
4	0	0	0	1	0	0	1
5	0	0	0	0	1	1	0
6	0	0	0	0	0	0	1



Incidence Matrix: Properties



	E ₁	E ₂	E ₃	E ₄	E ₅	E ₇	E ₈
1	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0
3	0	1	1	0	0	1	0
4	0	0	0	1	0	0	1
5	0	0	0	0	1	1	0
6	0	0	0	0	0	0	1

❖ Pros:

- Can represent **hypergraphs**
 - where one **edge** connects an **arbitrary** number of nodes

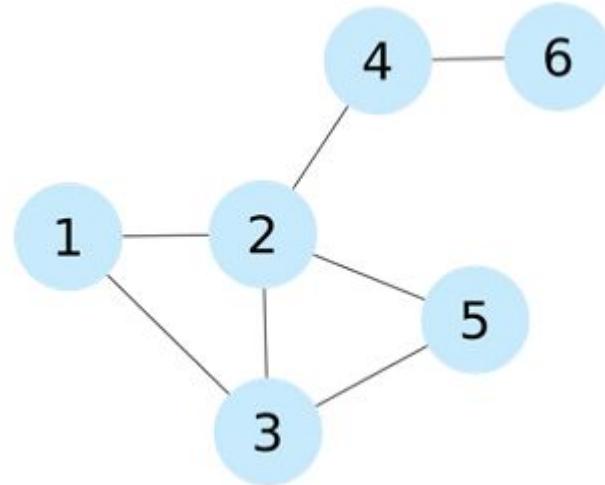
❖ Cons:

- Requires $n \times m$ bits (for most graphs $m \gg n$)



Data Structure: Laplacian Matrix

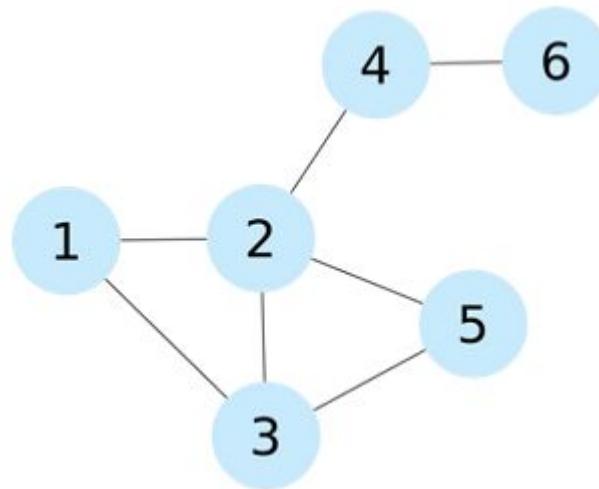
- ❖ Two-dimensional array of $n \times n$ integers
 - Similar structure to adjacency matrix
 - Diagonal of the Laplacian matrix indicates the degree of the node
 - L_{ij} is set to -1 if the two vertices i and j are connected, 0 otherwise



	1	2	3	4	5	6
1	2	-1	-1	0	0	0
2	-1	4	-1	-1	-1	0
3	-1	-1	3	0	-1	0
4	0	-1	0	2	0	-1
5	0	-1	-1	0	2	0
6	0	0	0	-1	0	1



Laplacian Matrix: Properties



All features of adjacency matrix

❖ Pros:

- Analyzing the graph structure by means of **spectral analysis**
 - Calculating **eigenvalues** of the matrix

	1	2	3	4	5	6
1	2	-1	-1	0	0	0
2	-1	4	-1	-1	-1	0
3	-1	-1	3	0	-1	0
4	0	-1	0	2	0	-1
5	0	-1	-1	0	2	0
6	0	0	0	-1	0	1



Basic Graph Algorithms

- ❖ Visiting all nodes:
 - Breadth-first Search (BFS)
 - Depth-first Search (DFS)
- ❖ Shortest path between two nodes
- ❖ Single-source shortest path problem
 - BFS (unweighted),
 - Dijkstra (nonnegative weights),
 - Bellman-Ford algorithm
- ❖ All-pairs shortest path problem
 - Floyd-Warshall algorithm



Improving Data Locality

- ❖ Performance of the **read/write** operations
 - Depends also on **physical organization** of the data
 - **Objective:** Achieve the best “data locality”
- ❖ **Spatial** locality:
 - if a data **item** has been **accessed**, the **nearby** data items are likely to be **accessed** in the following computations
 - e.g., during graph traversal
- ❖ **Strategy:**
 - in graph **adjacency matrix** representation, **exchange** rows and columns to improve the disk cache hit ratio
 - Specific **methods**: BFSL, Bandwidth of a Matrix, ...



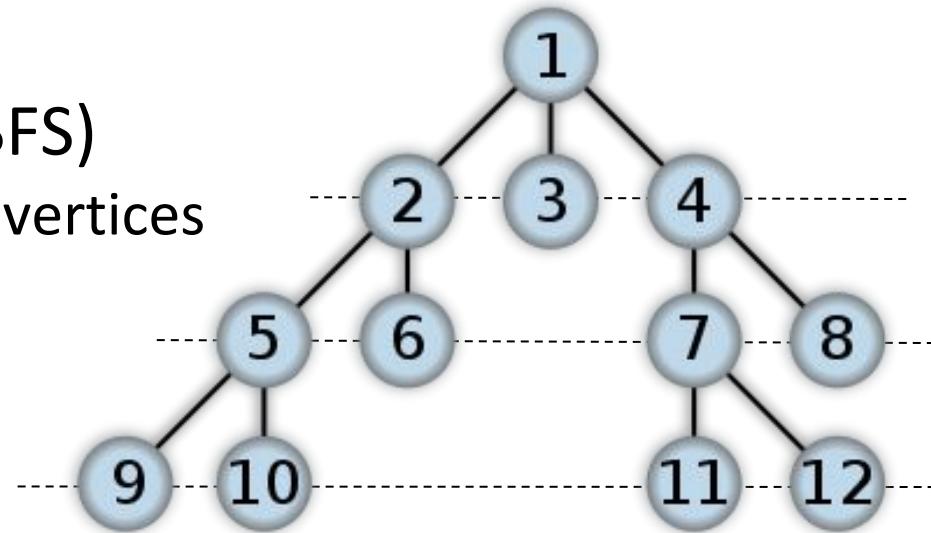
Breadth First Search Layout (BFSL)

- ❖ **Input:** vertices of a graph
- ❖ **Output:** a **permutation** of the vertices
 - with better cache performance for graph traversals
- ❖ **BFSL algorithm:**
 1. Select a **node** (at random, the origin of the traversal)
 2. **Traverse** the graph using the BFS alg.
 - generating a list of vertex identifiers in the **order** they are **visited**
 3. Take the **generated** list as the **new** vertices **permutation**



Breadth First Search Layout (2)

- ❖ Let us recall:
 - Breadth First Search (BFS)
 - FIFO **queue** of frontier vertices

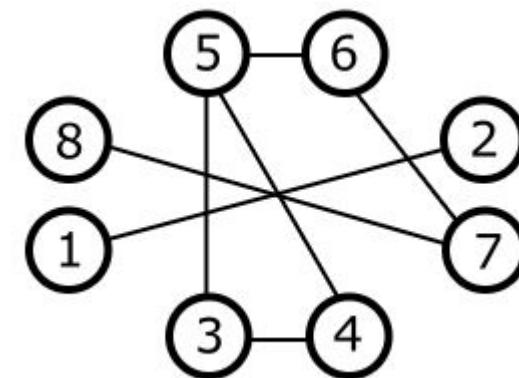
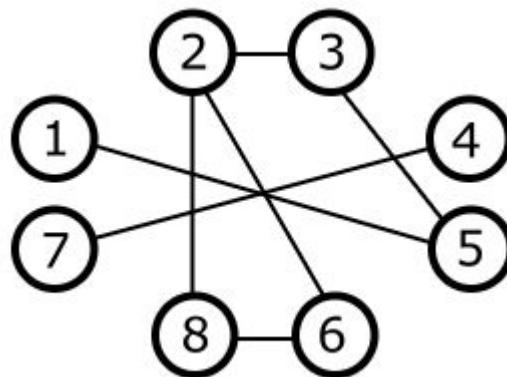


- ❖ Pros: **optimal** when starting from the **same node**
- ❖ Cons: starting from **other nodes**
 - The further, the worse



Matrix Bandwidth: Motivation

- Graph represented by adjacency matrix



$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$



Matrix Bandwidth: Formalization

- ❖ The minimum bandwidth problem
 - Bandwidth of a row in a matrix = the maximum distance between nonzero elements, where one is left of the diagonal and the other is right of the diagonal
 - Bandwidth of a matrix = maximum bandwidth of its rows
- ❖ Low bandwidth matrices are more cache friendly
 - Non zero elements (edges) clustered about the diagonal
- Bandwidth minimization problem: NP hard
 - For large matrices the solutions are only approximated



Graph Partitioning

- ❖ Some graphs are **too large** to be fully loaded into the **main memory** of a **single** computer
 - Usage of **secondary** storage **degrades** the **performance**
 - Scalable **solution**: **distribute** the graph on multiple nodes
- ❖ We need to **partition** the graph reasonably
 - Usually for a particular (set of) operation(s)
 - The shortest path, finding frequent patterns, **BFS**, spanning tree search
- ❖ This is **difficult** and graph DB are **often centralized**



Example: 1-Dimensional Partitioning

- ❖ Aim: **partitioning** the graph to solve BFS efficiently
 - Distributed into shared-nothing parallel system
 - Partitioning of the **adjacency matrix**
- ❖ **1D partitioning:**
 - Matrix **rows** are randomly assigned to the P nodes (processors) in the system
 - Each **vertex** and the **edges** emanating from it are **owned** by one processor



	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	1	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0



One-Dimensional Partitioning: BFS

- ❖ BSF with 1D partitioning
 1. Each **processor** has a set of vertices F (FIFO)
 2. The lists of neighbors of the vertices in F forms a set of **neighbouring vertices** N
 - Some owned by the current processor, some by others
 3. **Messages are sent** to all other processors... etc.
- ❖ 1D partitioning leads to **high messaging**
 - => **2D**-partitioning of adjacency matrix
 - ... lower messaging but **still very demanding**

Efficient **sharding** of a graph can be **difficult**



Types of Graph Databases

- ❖ **Single-relational** graphs
 - Edges are **homogeneous** in meaning
 - e.g., all edges represent friendship
- ❖ **Multi-relational (property)** graphs
 - Edges are **labeled by type**
 - e.g., friendship, business, communication
 - Vertices and edges maintain a **set** of key/value pairs
 - Representation of non-graphical data (**properties**)
 - e.g., name of a vertex, the weight of an edge



Graph Databases

- ❖ A graph database = a **set** of graphs

- ❖ Types of graph databases:
 - Transactional = **large set** of **small** graphs
 - e.g., chemical compounds, biological pathways, ...
 - Searching for graphs that match the query

 - Non-transactional = **few** numbers of **very large** graphs
 - or one huge (not connected) graph
 - e.g., Web graph, social networks, ...

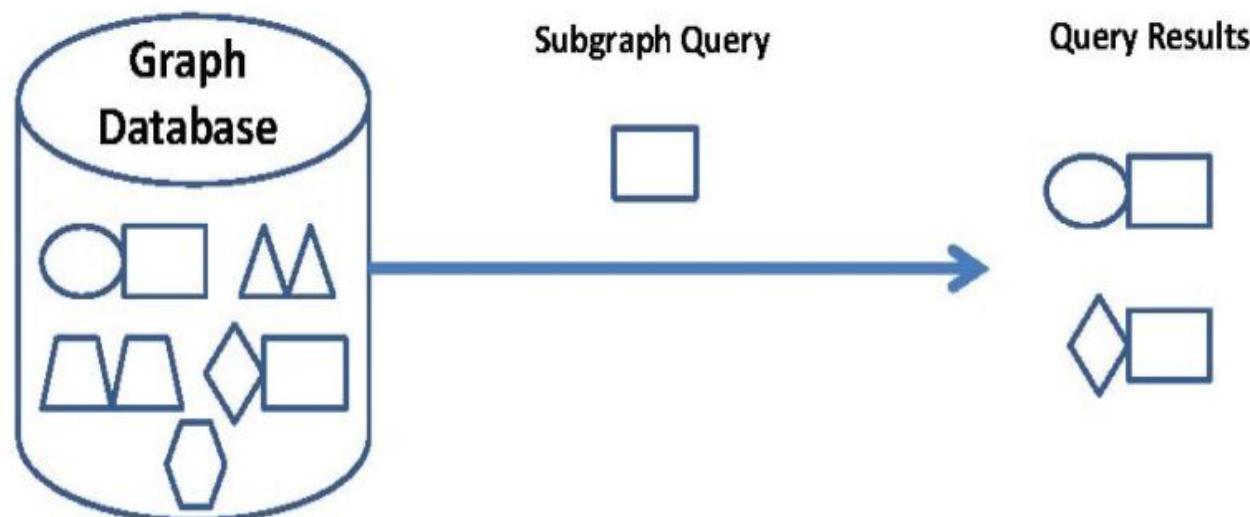


Transactional DBs: Queries

❖ Types of Queries

▪ Subgraph queries

- Search for a specific **pattern** in the graph database
- Query = a **small graph** or a graph, where some parts are uncertain
 - e.g., vertices with wildcard labels
- More **general** type: allow sub-graph **isomorphism**

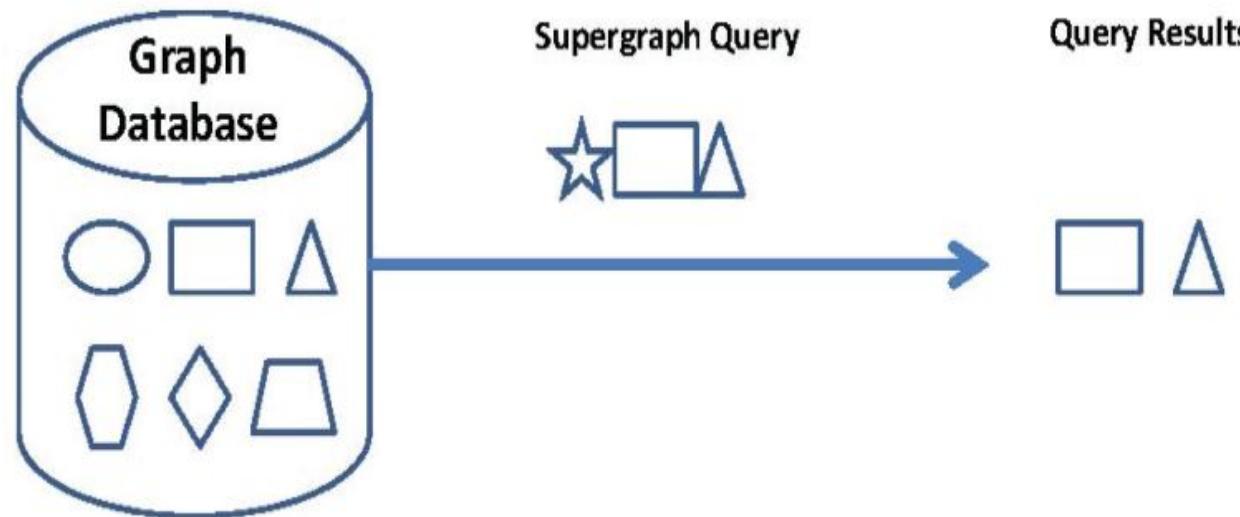




Transactional DBs: Queries (2)

- **Super-graph** queries

- Search for the graph **database members** whose whole structure is contained in the input **query**



- **Similarity (approximate matching)** queries

- Finds graphs which are **similar to** a given **query graph**
 - but not necessarily isomorphic
 - Key question: **how** to measure the **similarity**



Indexing & Query Evaluation

- ❖ Extract certain characteristics from each graph
 - And index these characteristics for each G_1, \dots, G_n
- ❖ Query evaluation in transactional graph DB
 1. Extraction of the characteristics from query graph q
 2. Filter the database (index) and identify a candidate set
 - Subset of the G_1, \dots, G_n graphs that should contain the answer
 3. Refinement - check all candidate graphs



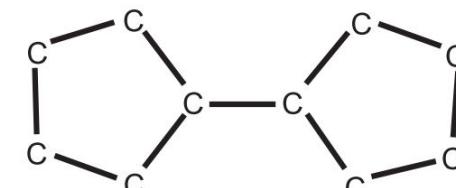
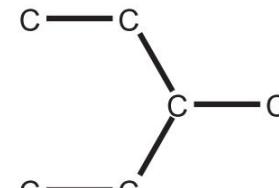
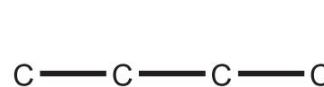
Subgraph Query Processing

1. **Mining-based Graph Indexing Techniques**
 - Idea: if **some features** of query graph q do not exist in data graph G , then G cannot contain q as its subgraph
 - Apply graph-mining methods to **extract** some **features** (sub-structures) from the graph database members
 - e.g., frequent sub-trees, frequent sub-graphs
 - An inverted **index** is created for each **feature**
2. **Non Mining-Based Graph Indexing Techniques**
 - Indexing of the **whole constructs** of the graph database
 - Instead of indexing only some selected features



Mining-based Technique

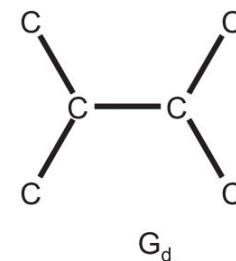
- ❖ Example method: GIndex [2004]
 - Indexing “frequent **discriminative** graphs”
 - Build **inverted** index for selected discriminative subgraphs



G_1

G_2

G_3

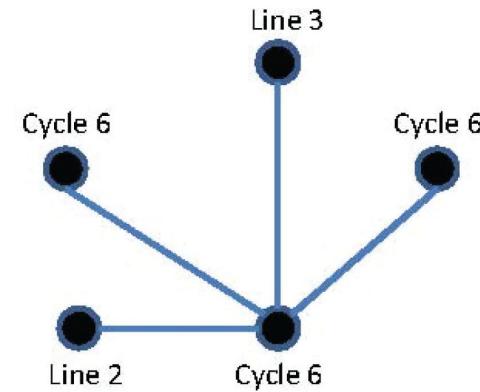
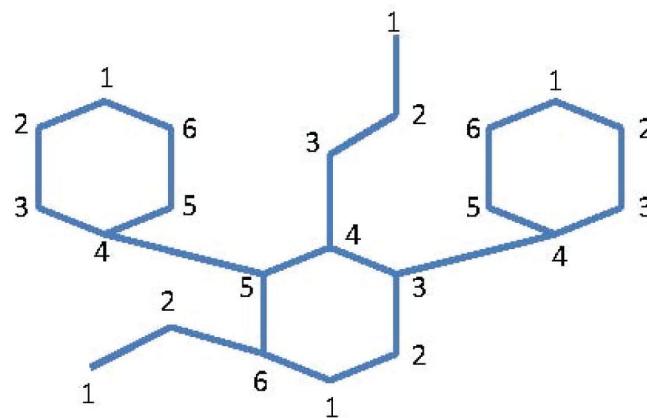




Non Mining-based Techniques

❖ Example: GString (2007)

- Model the graphs in the context of organic **chemistry** using basic structures
 - **Line** = series of vertices connected end to end
 - **Cycle** = series of vertices that form a close loop
 - **Star** = core vertex directly connects to several vertices





Non-transactional Graph Databases

- ❖ A **few** very **large** graphs
 - e.g., Web graph, social networks, ...
- ❖ Queries:
 - Nodes/edges with properties
 - Neighboring nodes/edges
 - Paths (all, shortest, etc.)

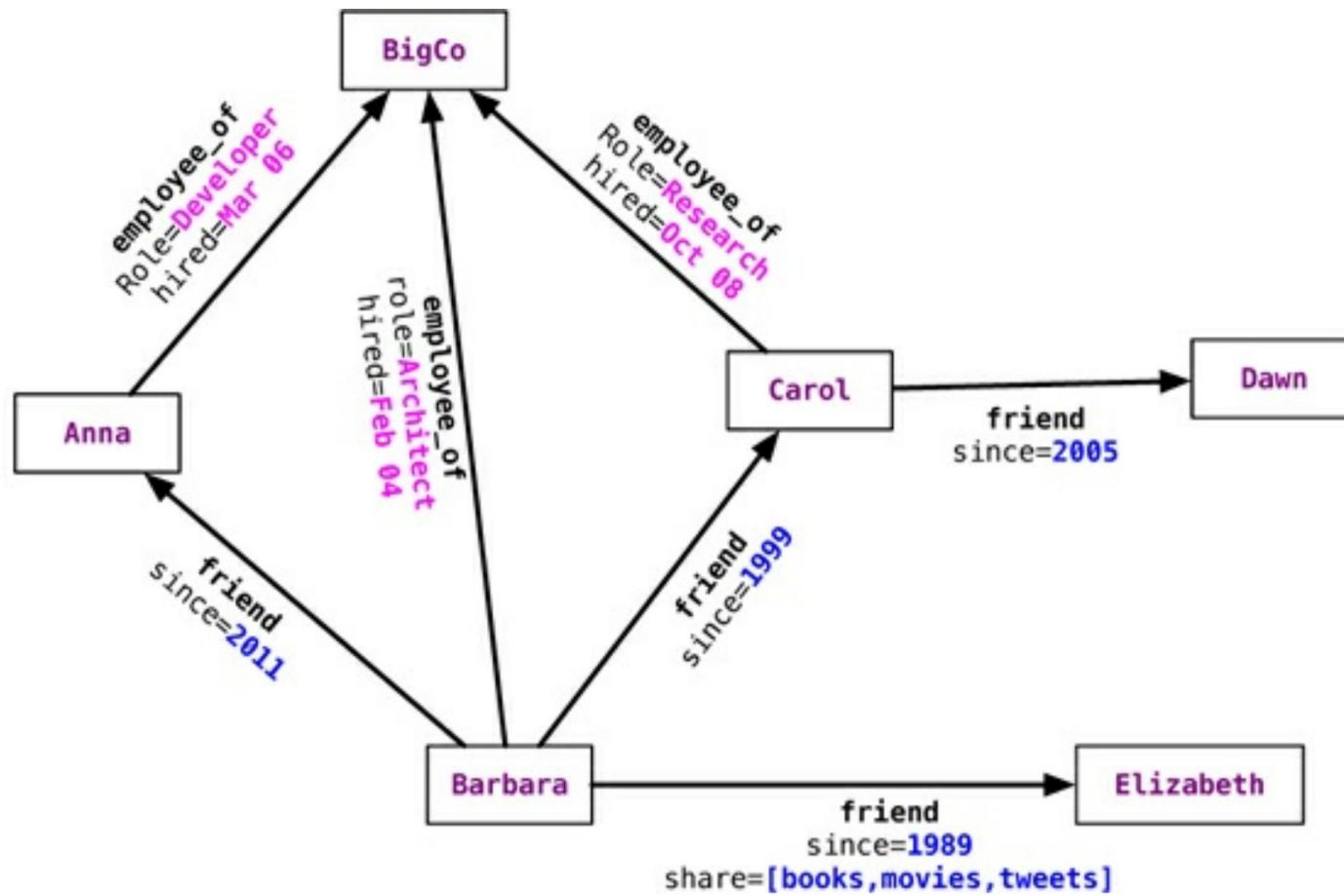


Basic Characteristics

- ❖ **Different** types of **relationships** between nodes
 - To represent **relationships** between **domain** entities
 - Or to model any kind of **secondary** relationships
 - Category, path, time-trees, spatial relationships, ...
- ❖ **No limit** to the number and kind of relationships
- ❖ **Relationships** have: type, start node, end node, own properties
 - e.g., “since when” did they become friends



Relationship Properties: Example





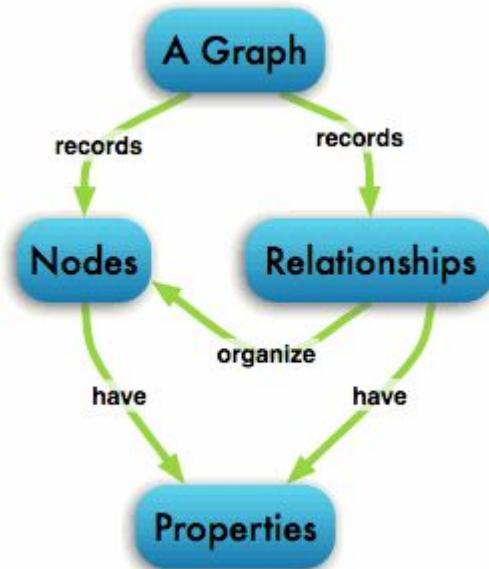
Graph DB vs. RDBMS

- ❖ **RDBMS** designed for a **single** type of **relationship**
 - “Think org charts”
 - Who works for who
 - Who is our lowest level common manager
- ❖ **Adding** a new relationship implies **schema changes**
 - New tables with foreign keys referencing other tables
- ❖ In RDBMS **we model** the graph **beforehand** based on the **traversal** we want
 - If the traversal changes, the data will have to change
 - **Graph DBs:** the relationship is not calculated but persisted



Neo4j: An exemplar Graph database

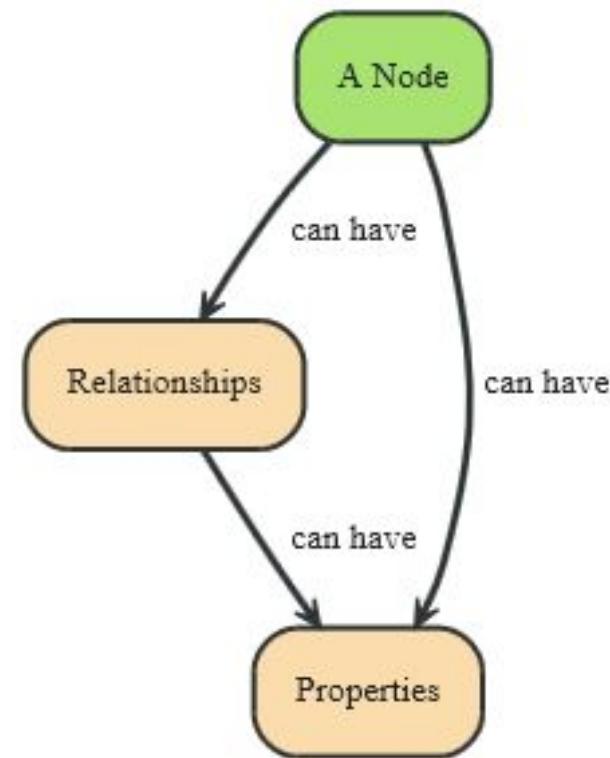
- ❖ Open source graph database
 - The most popular
- ❖ Initial release: 2007
- ❖ Written in: Java
- ❖ OS: cross-platform
- ❖ Stores data as nodes connected by directed, typed relationships
 - With properties on both
 - Called the “property graph”





Neo4j: Data Model

- ❖ Fundamental units: **nodes** + **relationships**
- ❖ Both can contain **properties**
 - **Key-value pairs**
 - Value can be of primitive type or an array of primitive type
 - **null** is **not a valid** property value
 - nulls can be modelled by the absence of a key

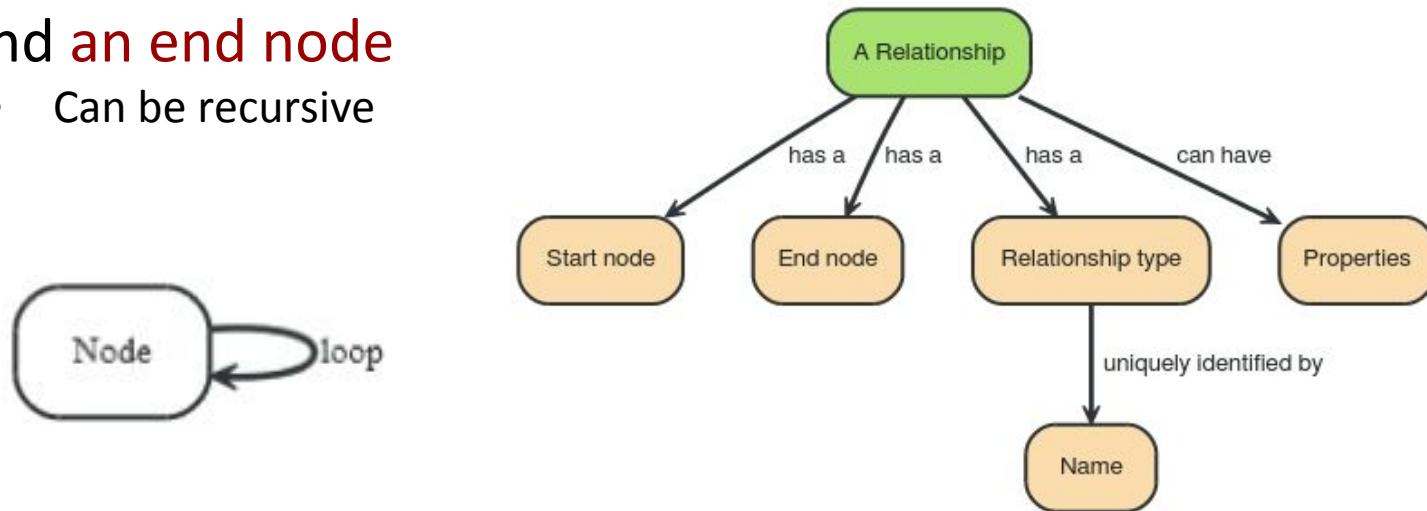




Data Model: Relationships

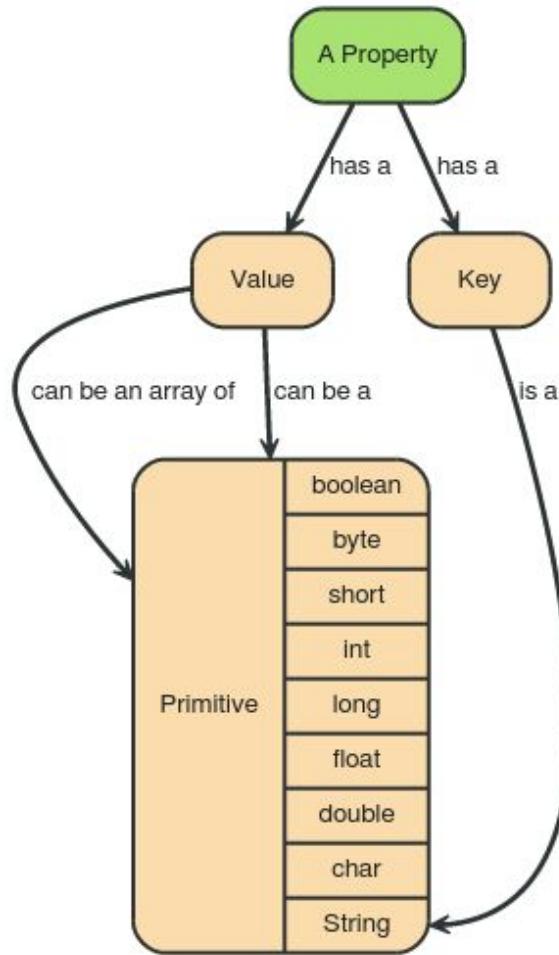
❖ Directed relationships (edges)

- Incoming and outgoing **edge**
 - Equally **efficient traversal** in both directions
 - Direction **can be ignored** if not needed by the application
- Always **a start** and **an end node**
 - Can be recursive





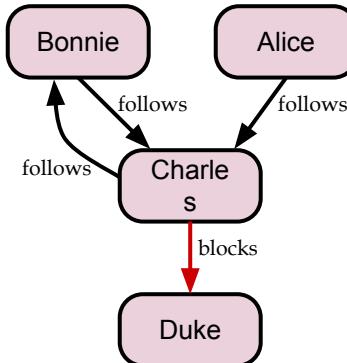
Data Model: Properties



Type	Description
boolean	true/false
byte	8-bit integer
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit IEEE 754 floating-point number
double	64-bit IEEE 754 floating-point number
char	16-bit unsigned integers representing Unicode characters
String	sequence of Unicode characters

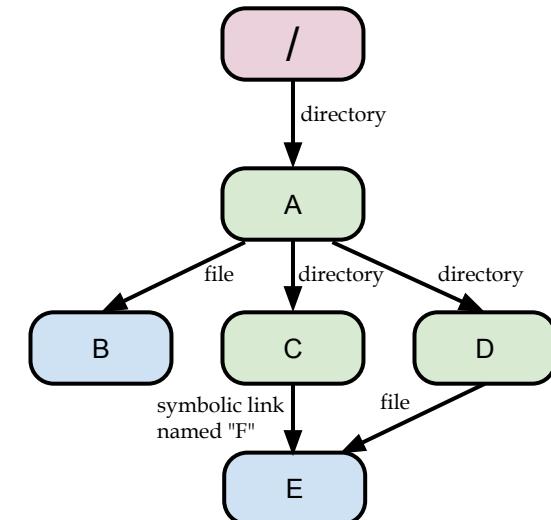


Examples



What	How
get who a person follows	outgoing <i>follows</i> relationships, depth one
get the followers of a person	incoming <i>follows</i> relationships, depth one
get who a person blocks	outgoing <i>blocks</i> relationships, depth one

What	How
get the full path of a file	incoming <i>file</i> relationships
get all paths for a file	incoming <i>file</i> and <i>symbolic link</i> relationships
get all files in a directory	outgoing <i>file</i> and <i>symbolic link</i> relationships, depth one
get all files in a directory, excluding symbolic links	outgoing <i>file</i> relationships, depth one
get all files in a directory, recursively	outgoing <i>file</i> and <i>symbolic link</i> relationships





Access to Neo4j

- ❖ Embedded database in Java system
- ❖ Language-specific connectors
 - Libraries to connect to a running Neo4j server
- ❖ Cypher query language
 - Standard language to query graph data
- ❖ HTTP REST API
- ❖ Gremlin graph traversal language (plugin)
- ❖ etc.



Native Java Interface: Example

```
Node alice = graphDb.createNode();
alice.setProperty("name", "Alice");
Node bonnie = graphDb.createNode();
bonnie.setProperty("name", "Bonnie");

Relationship a2b = alice.createRelationshipTo(bonnie,
FRIEND);
Relationship b2a = bonnie.createRelationshipTo(alice,
FRIEND);

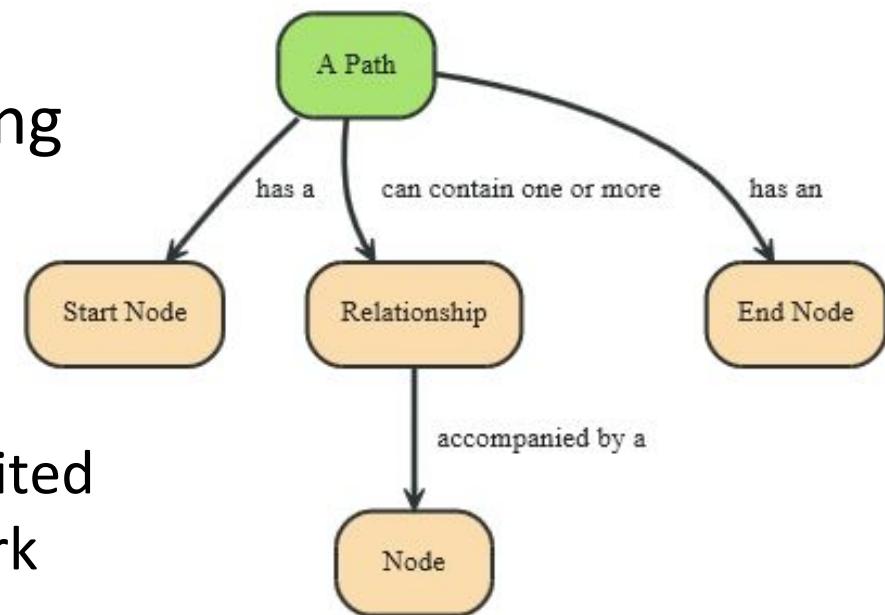
a2b.setProperty("quality", "a good one");
b2a.setProperty("since", 2003);
```

- ❖ **Undirected edge:**
 - Relationship between the nodes in **both directions**
 - **INCOMING** and **OUTGOING** relationships from a node



Data Model: Traversal + Path

- ❖ **Path** = one or more nodes + connecting relationships
 - Typically **retrieved as a result** of a query or a traversal
- ❖ **Traversing a graph** = visiting its nodes, following relationships according to some **rules**
 - Typically, a subgraph is visited
 - Neo4j: Traversal framework + Java API, Cypher, Gremlin





Traversal Framework

- ❖ A **traversal** is influenced by
 - Starting node(s) where the traversal will begin
 - Expanders – define what to traverse
 - i.e., relationship direction and type
 - Order – depth-first / breadth-first
 - Uniqueness – visit nodes (relationships, paths) only once
 - Evaluator – what to return and whether to stop or continue traversal beyond a current position

Traversal = TraversalDescription + **starting node(s)**



Traversal Framework – Java API

- ❖ org.neo4j...TraversalDescription
 - The main **interface** for defining **traversals**
 - Can specify branch ordering breadthFirst() / depthFirst()
- ❖ .relationships()
 - Adds the **relationship type** to traverse
 - e.g., traverse only edge types: FRIEND, RELATIVE
 - Empty (default) = traverse all relationships
 - Can also specify **direction**
 - Direction.BOTH
 - Direction.INCOMING
 - Direction.OUTGOING



Traversal Framework – Java API (2)

- ❖ org.neo4j...Evaluator
 - Used for deciding at each node: **should the traversal continue**, and should the node be included in the result
 - INCLUDE_AND_CONTINUE: Include this node in the result and continue the traversal
 - INCLUDE_AND_PRUNE: Include this node, do not continue traversal
 - EXCLUDE_AND_CONTINUE: Exclude this node, but continue traversal
 - EXCLUDE_AND_PRUNE: Exclude this node and do not continue
 - **Pre-defined evaluators:**
 - Evaluators.toDepth(int depth) /
Evaluators.fromDepth(int depth),
 - Evaluators.excludeStartPosition()
 - ...



Traversal Framework – Java API (3)

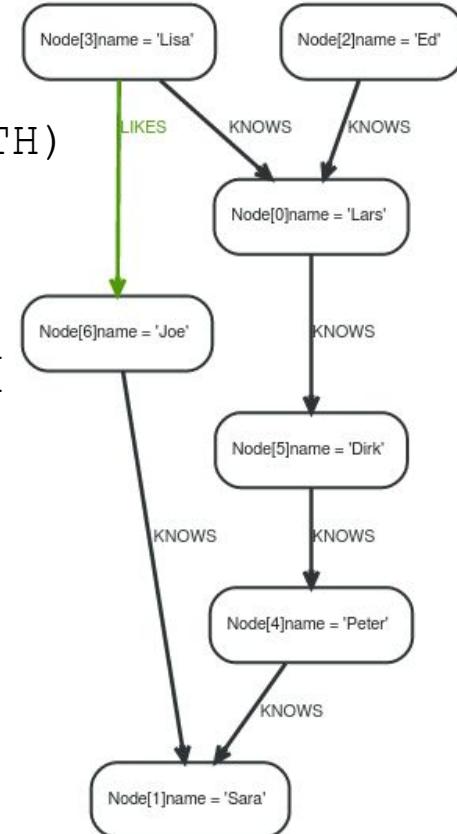
- ❖ org.neo4j...Uniqueness
 - Can be supplied to the TraversalDescription
 - Indicates under what circumstances a traversal may revisit the same position in the graph
- ❖ Traverser
 - Starts actual traversal given a TraversalDescription and starting node(s)
 - Returns an iterator over “steps” in the traversal
 - Steps can be: Path (default), Node, Relationship
 - The graph is actually traversed “lazily” (on request)



Example of Traversal

```
TraversalDescription desc =  
    db.traversalDescription()  
        .depthFirst()  
        .relationships(Rels.KNOWS, Direction.BOTH)  
        .evaluator(Evaluators.toDepth(3));  
  
// node is 'Ed' (Node[2])  
for (Node n : desc.traverse(node).nodes()) {  
    output += n.getProperty("name") + ", "  
}
```

Output: Ed, Lars, Lisa, Dirk, Peter,





Cypher Language

- ❖ Neo4j graph **query language**
 - For querying and updating
- ❖ **Declarative** – we say **what** we want
 - **Not how** to get it
 - **Not** necessary to express **traversals**
- ❖ **Human-readable**
- ❖ Inspired by SQL and SPARQL
- ❖ Still growing = syntax changes are often



Graph Database Summary

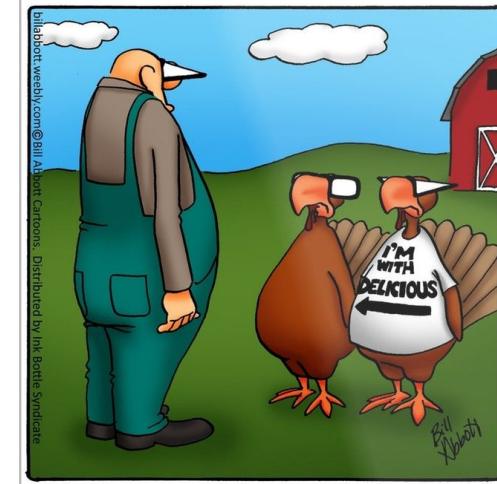
- ❖ Graph databases excel when objects are "indirectly" related to each other. Friends of friends, Cousins, your boss's boss's boss.
- ❖ Graph databases are suited for finding "structural patterns" in data.
 - If "X" buys "A", "B", "C" are they likely to buy "D"?
- ❖ When entities and their relationships are clustered



Next Time



- ❖ We finish up
- ❖ Alternate Final time:
 - You must have a documented conflict!
 - 9am on 12/9
- ❖ Remaining grading issues
 - See me next Tuesday





A Farewell to Files and Databases



Final Exam: 12/7 in SN014 and FB009
from 12pm-3pm

Open book, open notes, open-internet
No human communication

~12 questions Jupyter Notebook
~9 covering materials since the last midterm;
~3 comprehensive



Grading Status

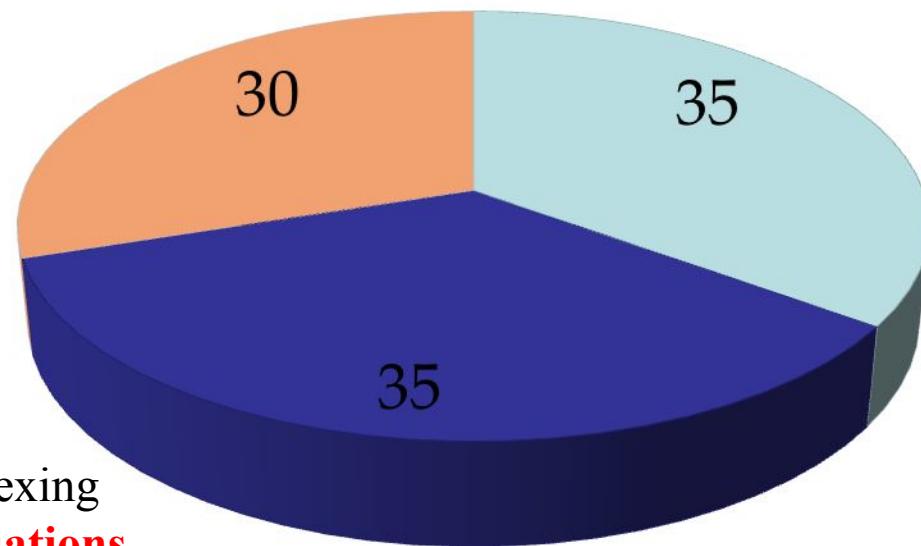
- Midterm
 - To my knowledge all issues are resolved and exams are graded
- Problem Sets (lowest score is dropped)
 - Everyone did well on Problem Set #6
 - Problem Set #5 graded soon! (today?)
 - All *issues* with other problem sets are resolved
 - Still needs grading (Done by Friday?)
- Exercises
 - You get 3 excused misses, the rest are 100% if turned in
- If you still have issues see me after class today or during my office hours tomorrow



Summary and What to study

- Relational Model
- **Out-of-core sorting**
- **Normal Forms**

Emphasis



- Database Indexing
- **Query Evaluations**
- **Query Optimization**
- **Transactions and Concurrency**

- Structured Query Language
- Integrating Dbases & programs
- **NoSQL**
 - **BASE, MapReduce, Hadoop**
 - **Document Model**
 - **Columnar Model**
 - **Graph Model**

- Applications
- Systems
- Foundations

