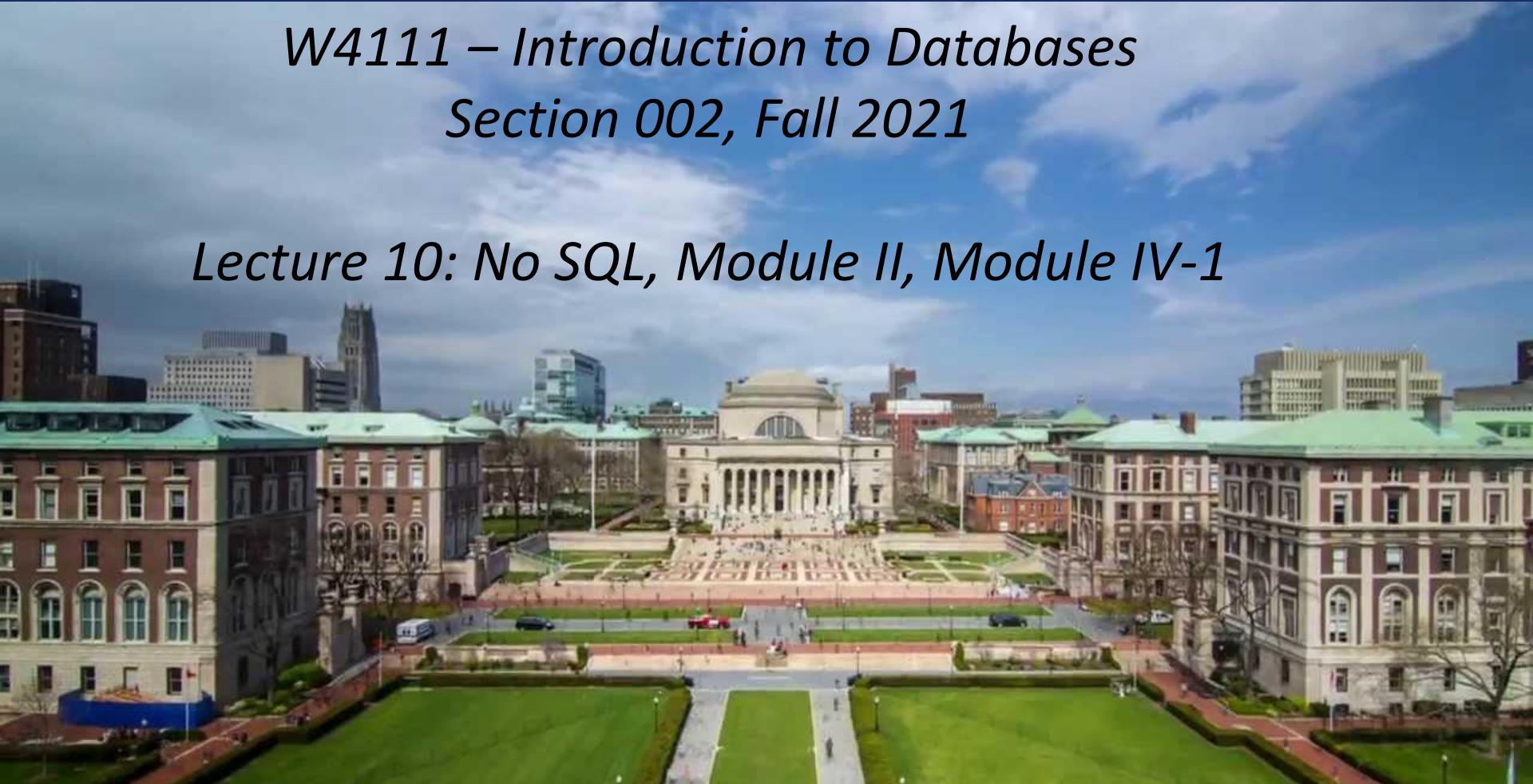


*W4111 – Introduction to Databases  
Section 002, Fall 2021*

*Lecture 10: No SQL, Module II, Module IV-1*



# *Contents*



# Contents

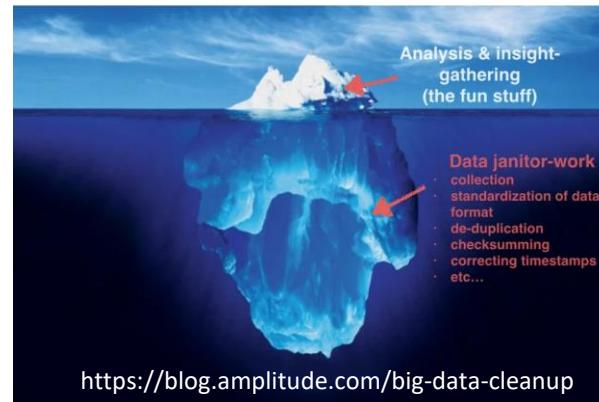
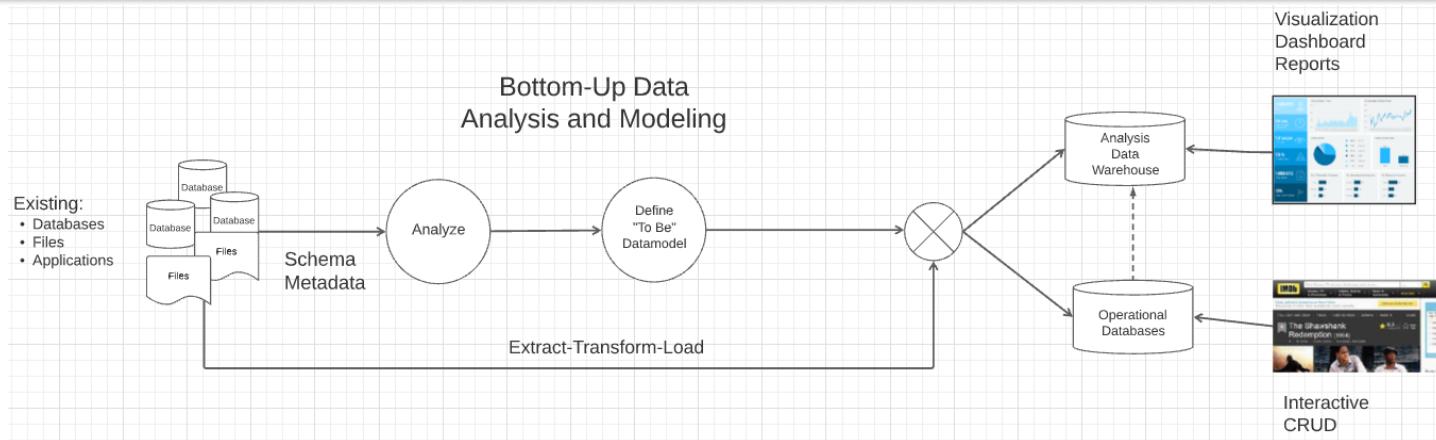
- Homework 3,4 and Concept
- Module II Continued
  - Indexes
  - Query processing and query optimization
- Normalization Part I: Introduction and Concepts
- Application Architectures:
  - Web applications
  - Data warehouse, data lake, decision support, ... ...

# *HW3A – Setting the Foundation for HW3 and HW4 → Project*

# Types of Database Application

- Overly simplistically, there are two broad categories of database applications.
  1. Online/Interactive/Transactional:
    - Supports users interactively performing CRUD on objects-entities:
      - Retrieve: Support for querying and finding entities using relatively simple, often predefined parameterized queries.
      - Create, Update, Delete of specific entities
    - Examples:
      - Online banking, e-commerce, ...
      - SSOL
  2. Decision Support:
    - Read only.
    - Ad-hoc, complex queries.
    - Exploration, visualization, report generation, ... ...
- Most environments have both types, which are integrated.
- Both types require well-designed databases and data models.

# Homework 3A – Laying the Foundation



# *A Whirlwind Tour of Indexes*

## *(Database Systems Concepts, V7, Ch. 14)*



# Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



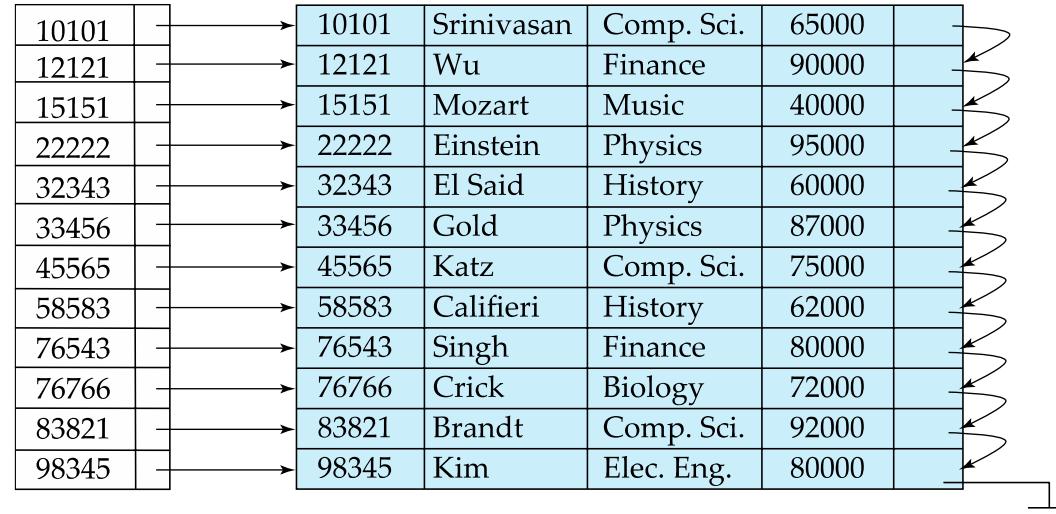
# Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering index**.
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.



# Dense Index Files

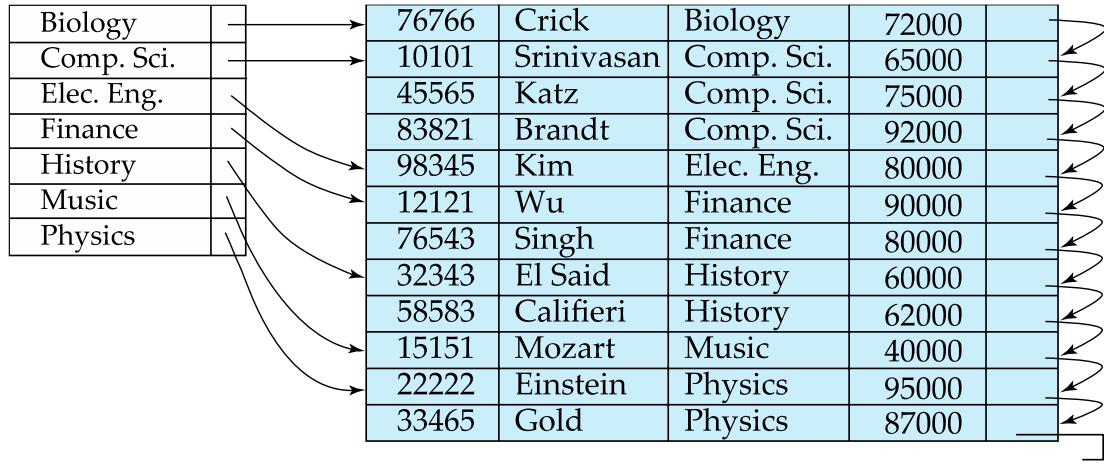
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





## Dense Index Files (Cont.)

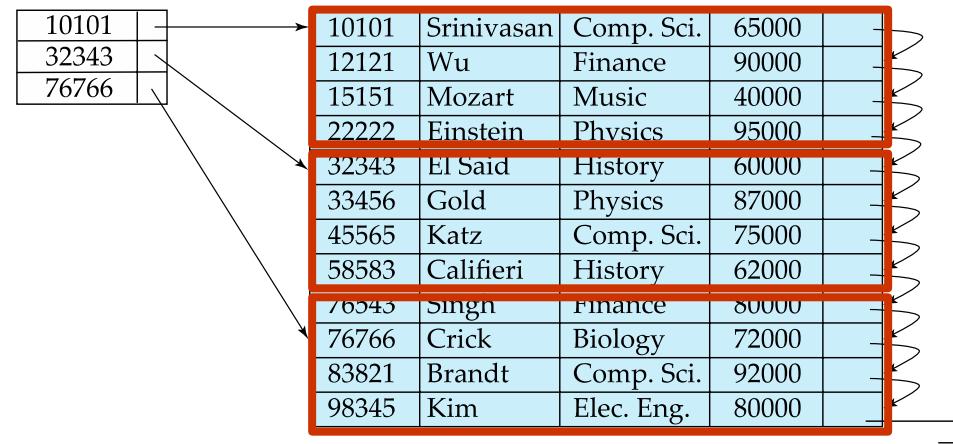
- Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*





# Sparse Index Files

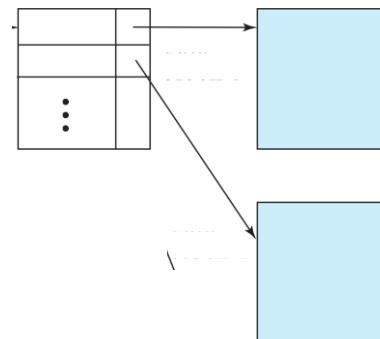
- **Sparse Index:** contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points





# Sparse Index Files (Cont.)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff:**
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

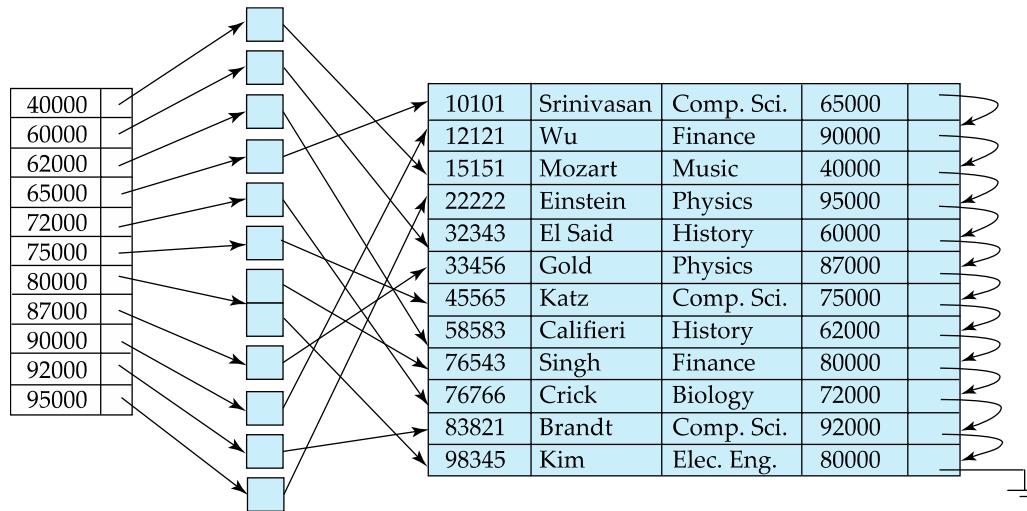


- For unclustered index: sparse index on top of dense index (multilevel index)



# Secondary Indices Example

- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

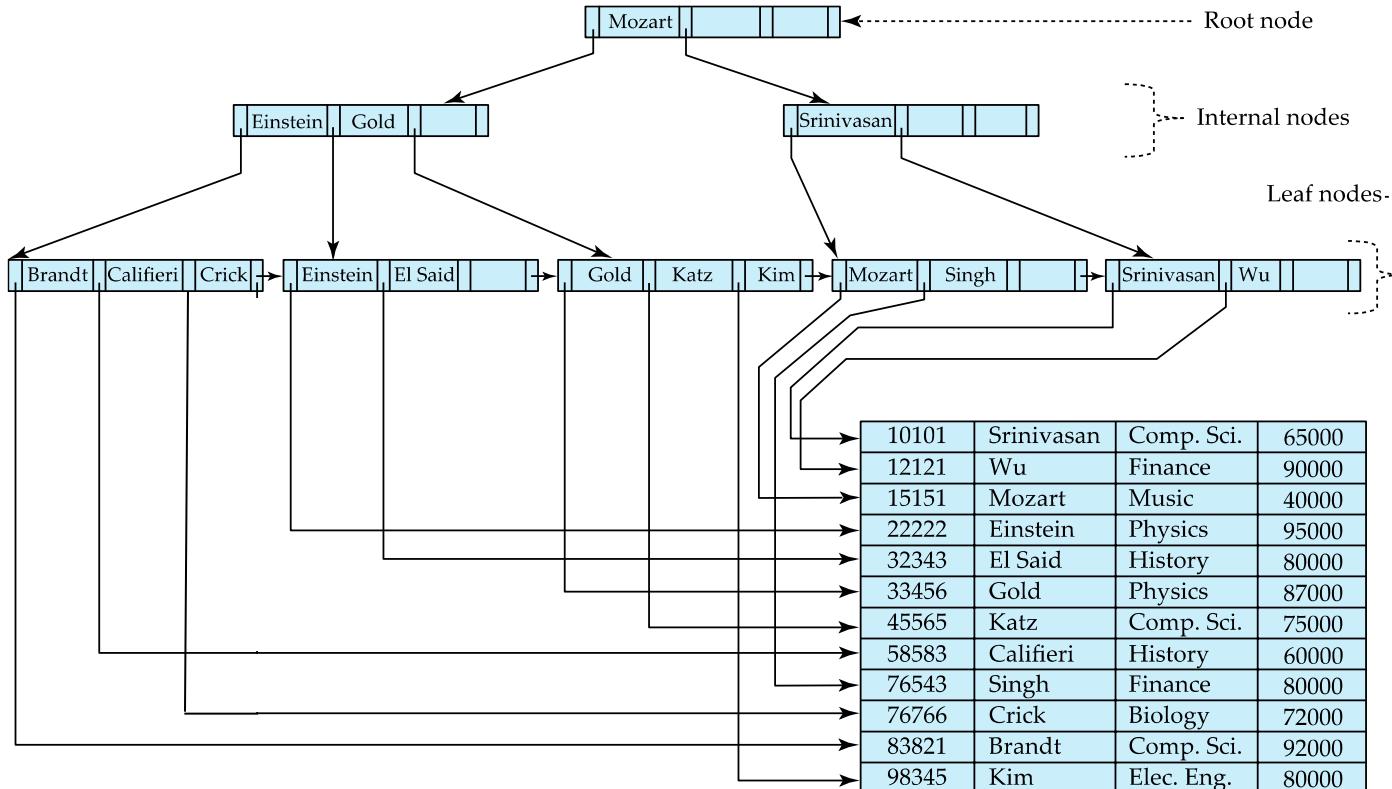


# Indices on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name*, *ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name*, *ID*)
- (nameLast, nameFirst, birthyear)
  - nameLast [nameLast = “Ferguson”] [nameLast like “Fer%”]
  - nameLast, nameFirst
  - nameLast, nameFirst, birthyear
- NOT and index on
  - nameFirst, nameLast
  - birthyear
  - nameLast like [%er%]



# Example of B+-Tree





## B<sup>+</sup>-Tree Index Files (Cont.)

A B<sup>+</sup>-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between  $\lceil n/2 \rceil$  and  $n$  children.
- A leaf node has between  $\lceil (n-1)/2 \rceil$  and  $n-1$  values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and  $(n-1)$  values.



# B<sup>+</sup>-Tree Node Structure

- Typical node

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

- $K_i$  are the search-key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

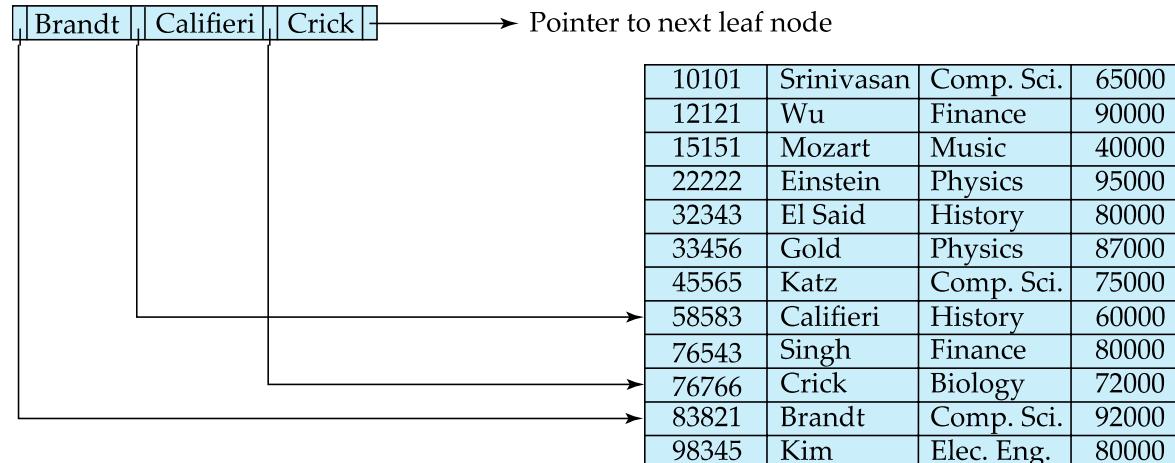
(Initially assume no duplicate keys, address duplicates later)



# Leaf Nodes in B+-Trees

Properties of a leaf node:

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_j$
- If  $L_i, L_j$  are leaf nodes and  $i < j$ ,  $L_i$ 's search-key values are less than or equal to  $L_j$ 's search-key values
- $P_n$  points to next leaf node in search-key order  
leaf node





# Non-Leaf Nodes in B<sup>+</sup>-Trees

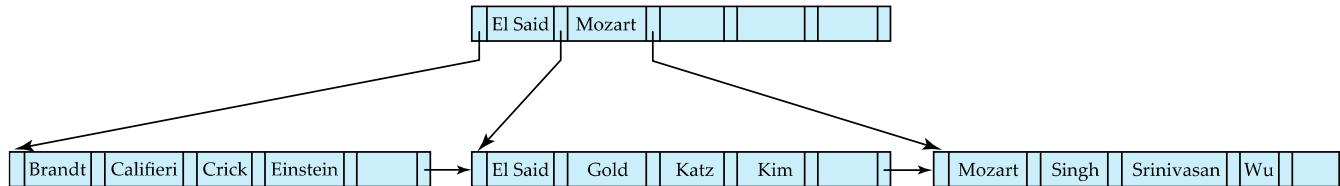
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with  $m$  pointers:
  - All the search-keys in the subtree to which  $P_1$  points are less than  $K_1$
  - For  $2 \leq i \leq n - 1$ , all the search-keys in the subtree to which  $P_i$  points have values greater than or equal to  $K_{i-1}$  and less than  $K_i$
  - All the search-keys in the subtree to which  $P_n$  points have values greater than or equal to  $K_{n-1}$
  - General structure

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------



## Example of B<sup>+</sup>-tree

- B<sup>+</sup>-tree for *instructor* file ( $n = 6$ )



- Leaf nodes must have between 3 and 5 values ( $\lceil (n-1)/2 \rceil$  and  $n-1$ , with  $n = 6$ ).
- Non-leaf nodes other than root must have between 3 and 6 children ( $\lceil (n/2) \rceil$  and  $n$  with  $n = 6$ ).
- Root must have at least 2 children.



# Observations about B+-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
  - Level below root has at least  $2 * \lceil n/2 \rceil$  values
  - Next level has at least  $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$  values
  - .. etc.
  - If there are  $K$  search-key values in the file, the tree height is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

# Show the Simulator

<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>



# Hashing



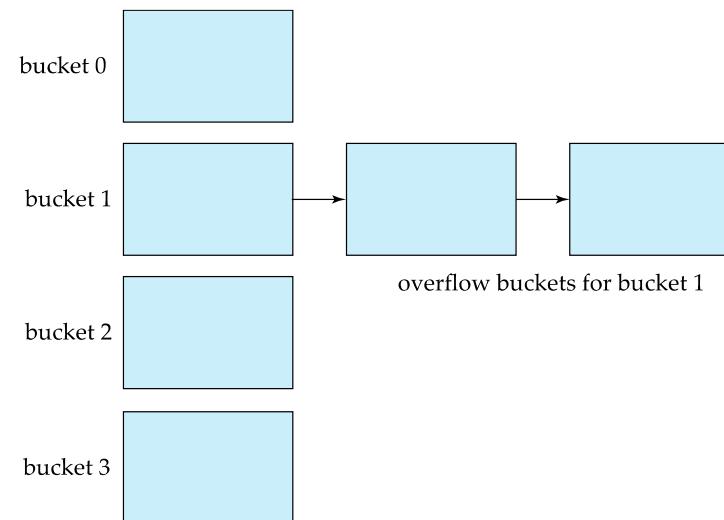
# Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$ .
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records



## Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing** (also called **closed hashing** or **open hashing** depending on the book you use)
  - An alternative, called **open addressing** (also called **open hashing** or **closed hashing** depending on the book you use) which does not use overflow buckets, is not suitable for database applications.





# Example of Hash File Organization

Hash file organization of *instructor* file, using *dept\_name* as key.

bucket 0


bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7




# Deficiencies of Static Hashing

- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

# Show the Simulator

<http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html>

<https://opendsa-server.cs.vt.edu/ODSA/AV/Development/hashAV.html>

nameLast=“Ferguson”

nameLast >= “Ferguson” and nameLast <= “Guthrie”

Select \* from professors join students using (uni)

$$O(N)*O(M) \rightarrow O(N*M)$$

$$O(N)+O(M)+O(1)*O(N) \rightarrow O(N+M)$$

# *Query Processing*

# Query Processing Overview

# Query Compilation

## Preview of Query Compilation

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

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

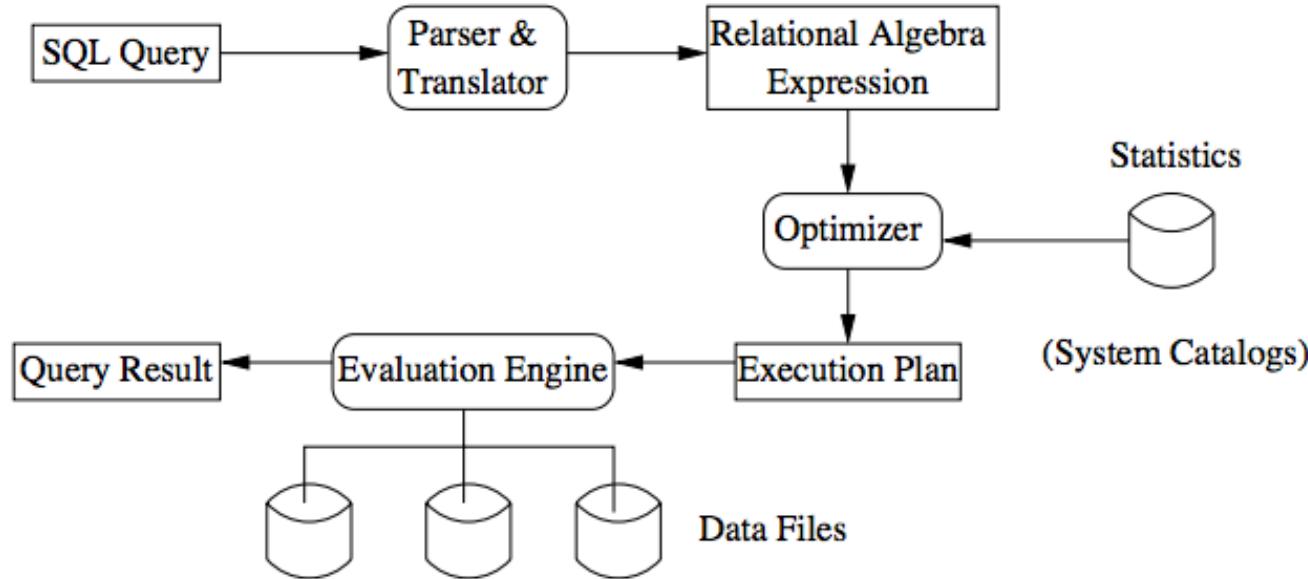
- a) *Parsing.* A *parse tree* for the query is constructed.
- b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.
- c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

# Parsing and Execution

- Parser/Translator
  - Verifies syntax correctness and generates a *parse tree*.
  - Converts to *logical plan tree* that defines how to execute the query.
    - Tree nodes are *operator(tables, parameters)*
    - Edges are the flow of data “up the tree” from node to node.
- Optimizer
  - Modifies the logical plan to define an improved execution.
  - Query rewrite/transformation.
  - Determines *how* to choose among multiple implementations of operators.
- Engine
  - Executes the plan
  - May modify the plan to *optimize* execution, e.g. using indexes.

# Query Processing Overview

## Basic Steps in Processing an SQL Query



# Chapter 15

## From the Book



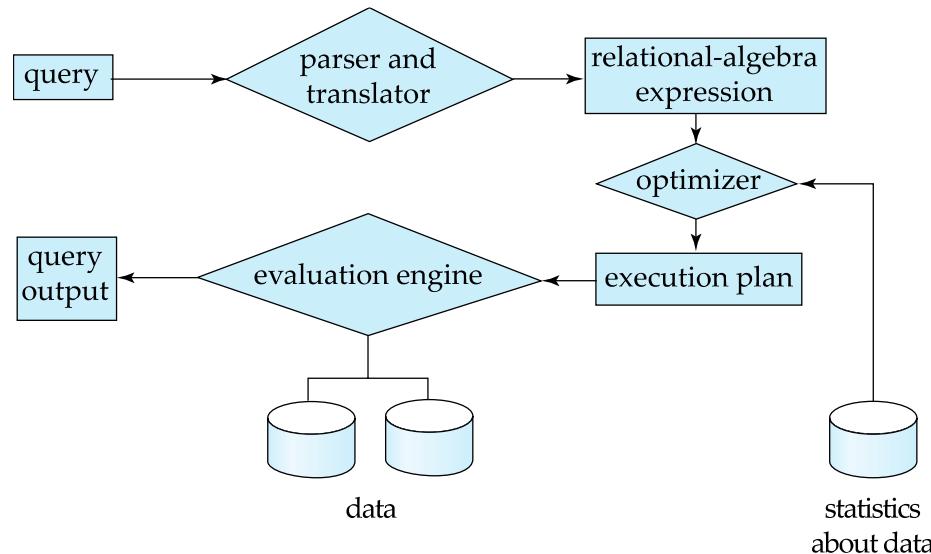
# Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions



# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





# Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



# Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g.,  $\sigma_{\text{salary} < 75000}(\Pi_{\text{salary}}(\text{instructor}))$  is equivalent to  $\Pi_{\text{salary}}(\sigma_{\text{salary} < 75000}(\text{instructor}))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
  - Use an index on *salary* to find instructors with  $\text{salary} < 75000$ ,
  - Or perform complete relation scan and discard instructors with  $\text{salary} \geq 75000$



# Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g.. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 16
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost



# Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU, and network communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk



# Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks \* average-seek-cost
  - Number of blocks read \* average-block-read-cost
  - Number of blocks written \* average-block-write-cost
- For simplicity we just use the **number of block transfers** from disk and the **number of seeks** as the cost measures
  - $t_T$  – time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - $t_S$  – time for one seek
  - Cost for  $b$  block transfers plus  $S$  seeks
$$b * t_T + S * t_S$$
- $t_S$  and  $t_T$  depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk:  $t_S = 4$  msec and  $t_T = 0.1$  msec
  - SSD:  $t_S = 20\text{-}90$  microsec and  $t_T = 2\text{-}10$  microsec for 4KB



# Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice



# Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000    *takes*: 10,000
  - Number of blocks of *student*: 100    *takes*: 400
- $R \text{ JOIN } L = L \text{ JOIN } R$ 
  - R is scan table
  - L is probe



# Nested-Loop Join

- To compute the theta join  $r \bowtie_{\theta} s$   
**for each tuple  $t_r$  in  $r$  do begin**  
    **for each tuple  $t_s$  in  $s$  do begin**  
        test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$   
        if they do, add  $t_r \bullet t_s$  to the result.  
    **end**  
**end**
- $r$  is called the **outer relation** and  $s$  the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.



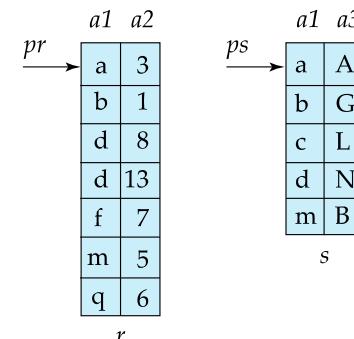
# Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple  $t_r$  in the outer relation  $r$ , use the index to look up tuples in  $s$  that satisfy the join condition with tuple  $t_r$ .
- Worst case: buffer has space for only one page of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r(t_T + t_S) + n_r * c$ 
  - Where  $c$  is the cost of traversing index and fetching all matching  $s$  tuples for one tuple of  $r$
  - $c$  can be estimated as cost of a single selection on  $s$  using the join condition.
- If indices are available on join attributes of both  $r$  and  $s$ , use the relation with fewer tuples as the outer relation.



# Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
  1. Join step is similar to the merge stage of the sort-merge algorithm.
  2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  3. Detailed algorithm in book





## Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:  
 $b_r + b_s$  block transfers +  $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$  seeks  
+ the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B<sup>+</sup>-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B<sup>+</sup>-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

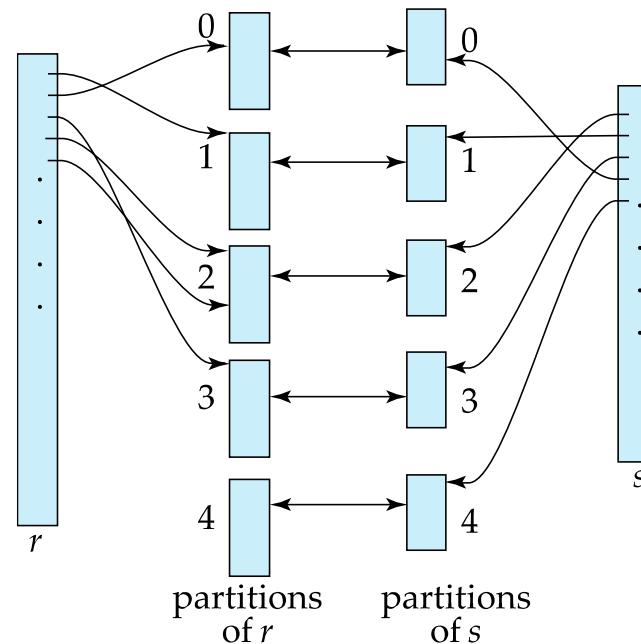


# Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function  $h$  is used to partition tuples of both relations
- $h$  maps  $JoinAttrs$  values to  $\{0, 1, \dots, n\}$ , where  $JoinAttrs$  denotes the common attributes of  $r$  and  $s$  used in the natural join.
  - $r_0, r_1, \dots, r_n$  denote partitions of  $r$  tuples
    - Each tuple  $t_r \in r$  is put in partition  $r_i$  where  $i = h(t_r[JoinAttrs])$ .
  - $r_0, r_1, \dots, r_n$  denotes partitions of  $s$  tuples
    - Each tuple  $t_s \in s$  is put in partition  $s_i$ , where  $i = h(t_s[JoinAttrs])$ .
- Note: In book, Figure 12.10  $r_i$  is denoted as  $H_{ri}$ ,  $s_i$  is denoted as  $H_{si}$  and  $n$  is denoted as  $n_h$ .
- $R \text{ JOIN } L \rightarrow O(R^*L)$
- 1. Build a hash index on  $L \rightarrow O(L)$
- $O(R) * O(1) + O(L)$



## Hash-Join (Cont.)





# Hash-Join Algorithm

The hash-join of  $r$  and  $s$  is computed as follows.

1. Partition the relation  $s$  using hashing function  $h$ . When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition  $r$  similarly.
3. For each  $i$ :
  - (a) Load  $s_i$  into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one  $h$ .
  - (b) Read the tuples in  $r_i$  from the disk one by one. For each tuple  $t_r$ , locate each matching tuple  $t_s$  in  $s_i$  using the in-memory hash index. Output the concatenation of their attributes.

Relation  $s$  is called the **build input** and  $r$  is called the **probe input**.



## Hash-Join algorithm (Cont.)

- The value  $n$  and the hash function  $h$  is chosen such that each  $s_i$  should fit in memory.
  - Typically  $n$  is chosen as  $\lceil b_s/M \rceil * f$  where  $f$  is a “**fudge factor**”, typically around 1.2
  - The probe relation partitions  $s_i$  need not fit in memory
- **Recursive partitioning** required if number of partitions  $n$  is greater than number of pages  $M$  of memory.
  - instead of partitioning  $n$  ways, use  $M - 1$  partitions for  $s$
  - Further partition the  $M - 1$  partitions using a different hash function
  - Use same partitioning method on  $r$
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB



# Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection:**
  - perform projection on each tuple
  - followed by duplicate elimination.



# Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization: **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end



# Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization:** generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining:** pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail

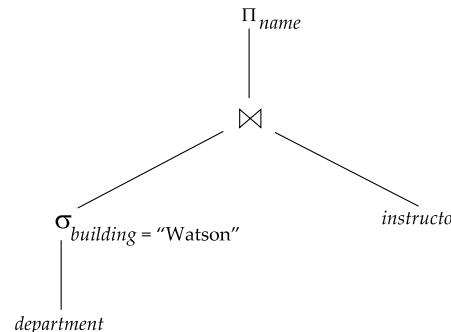


# Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building = "Watson"}(department)$$

then compute the store its join with *instructor*, and finally compute the projection on *name*.





# Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering:** use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time



# Pipelining

- **Pipelined evaluation:** evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of
$$\sigma_{building = "Watson"}(department)$$
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**



# Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain “**state**” so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining



# Pipelining (Cont.)

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g., file scan: initialize file scan
        - state: pointer to beginning of file
      - E.g., merge join: sort relations;
        - state: pointers to beginning of sorted relations
    - **next()**
      - E.g., for file scan: Output next tuple, and advance and store file pointer
      - E.g., for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**

# *Normalization*



# Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in\_dep*, which represents the natural join on the relations *instructor* and *department*

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

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

# Evil's of Redundancy



## *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?



# Decomposition

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

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

into

*employee1* (*ID*, *name*)

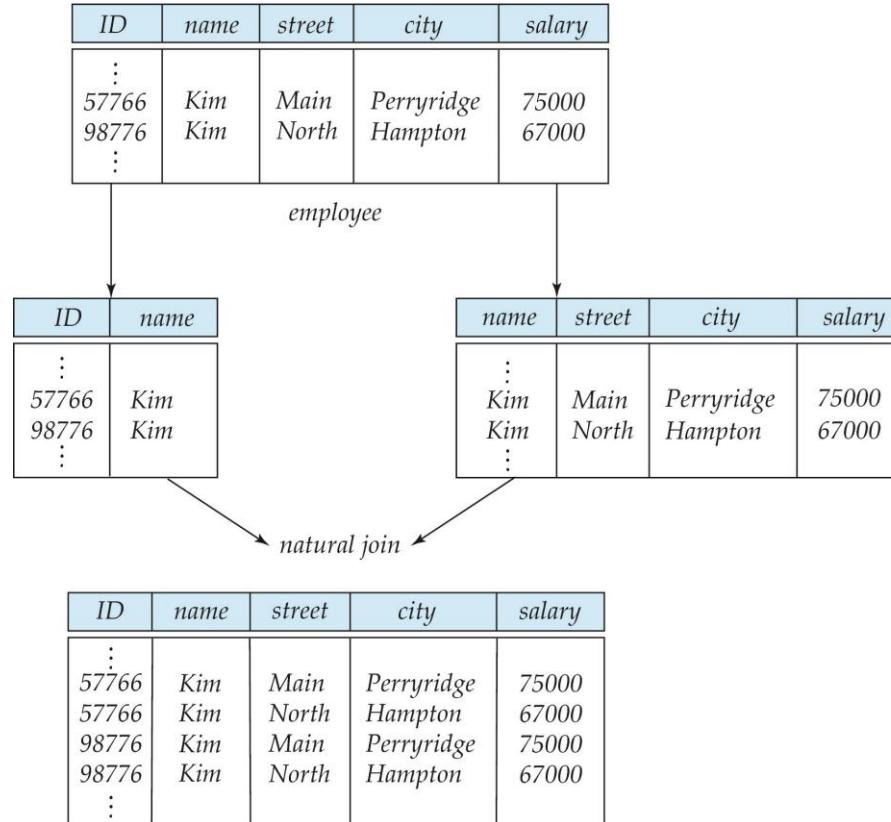
*employee2* (*name*, *street*, *city*, *salary*)

The problem arises when we have two employees with the same name

- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.



# A Lossy Decomposition





# Lossless Decomposition

- Let  $R$  be a relation schema and let  $R_1$  and  $R_2$  form a decomposition of  $R$ . That is  $R = R_1 \cup R_2$
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing  $R$  with the two relation schemas  $R_1 \cup R_2$
- Formally,

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

- And, conversely a decomposition is lossy if

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



# Example of Lossless Decomposition

- Decomposition of  $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

A	B	C
$\alpha$	1	A
$\beta$	2	B

$r$

A	B
$\alpha$	1
$\beta$	2

$\Pi_{A,B}(r)$

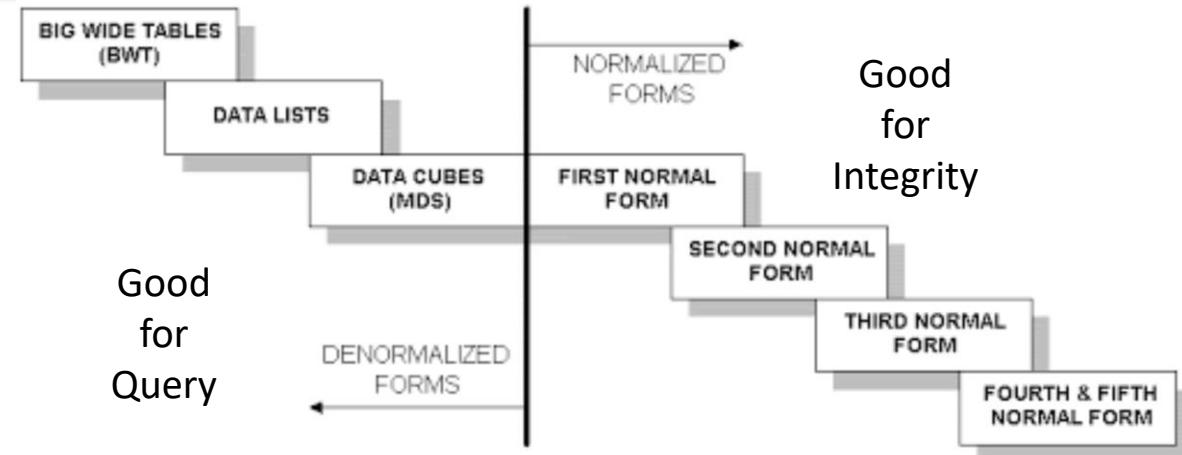
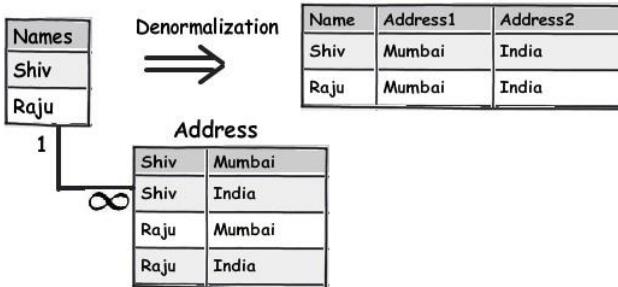
B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
$\alpha$	1	A
$\beta$	2	B

# Wide Flat Tables



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



# Normalization Theory

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

We will cover details in a future lecture.

# *Data Models and REST*

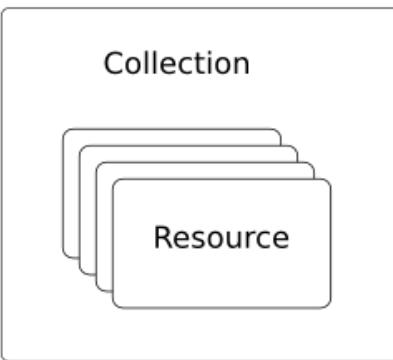
# Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains***.
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

# REST and Resources

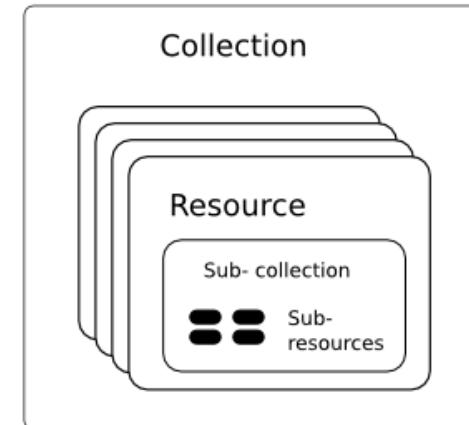
## Resource Model



A Collection with  
Resources

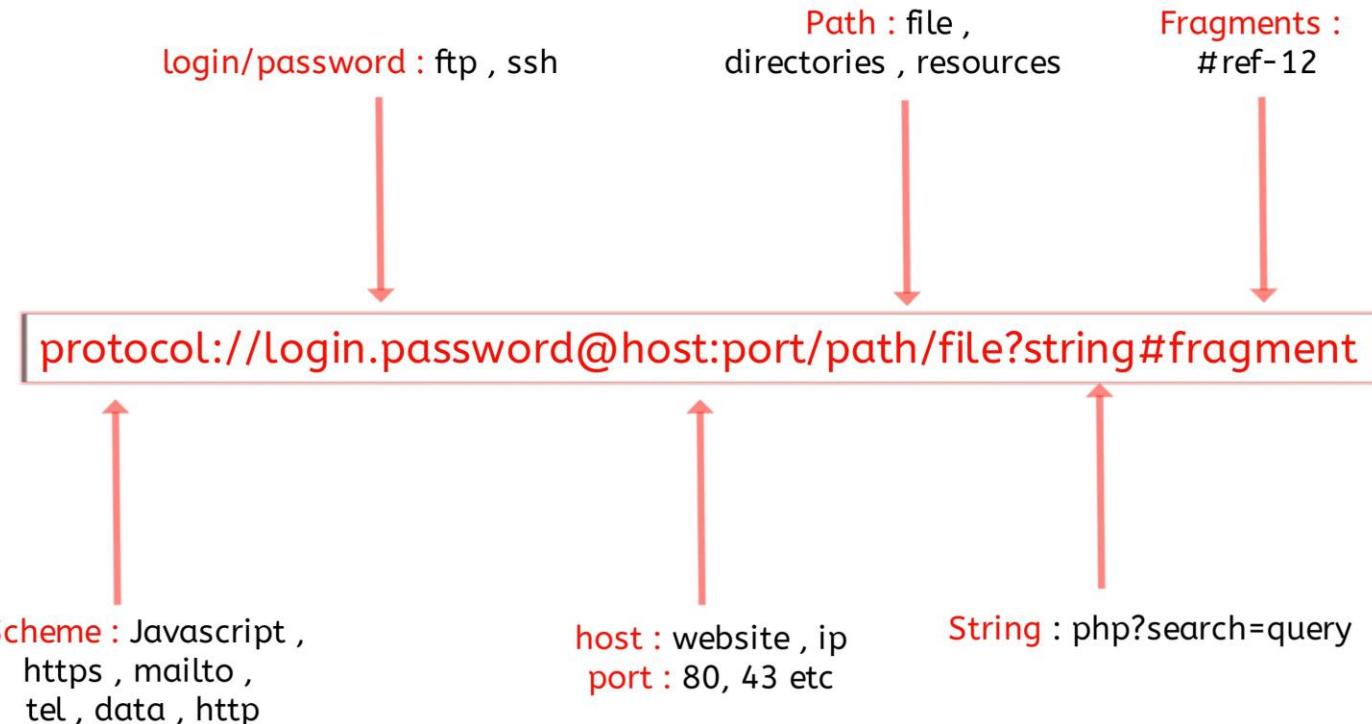


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



# URL Mappings

- Some URLs: This gets you to the database service (program)
  - <http://127.0.0.1:5001/api>
  - mysql+pymysql://dbuser:dbuser@localhost
  - mongodb://mongodb0.example.com:27017
  - neo4j://graph.example.com:7687
- You still have to get into a database within the service:
  - SQL: use lahmansbaseballdb
  - MongoDB: db.lahmansbaseballdb
  - <HTTP://127.0.0.1:5001/api/lahmansbaseballdb>
  - ... ...
- And then into things inside of things inside of things ... In the database.

# Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...;  SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

# PUT, DELETE, UPDATE

- /people?
  - POST (INSERT)
  - GET (SELECT ... WHERE ...)
- /people/21
  - WHERE peopleID=21
  - DELETE → DELETE WHERE
  - PUT → UPDATE SET ..... WHERE
  - GET SELECT ... WHERE

# Simplistic, Conceptual Mapping (Examples)

POST ▼ http://127.0.0.1:5001/api/people/willite01/batting Send ▼

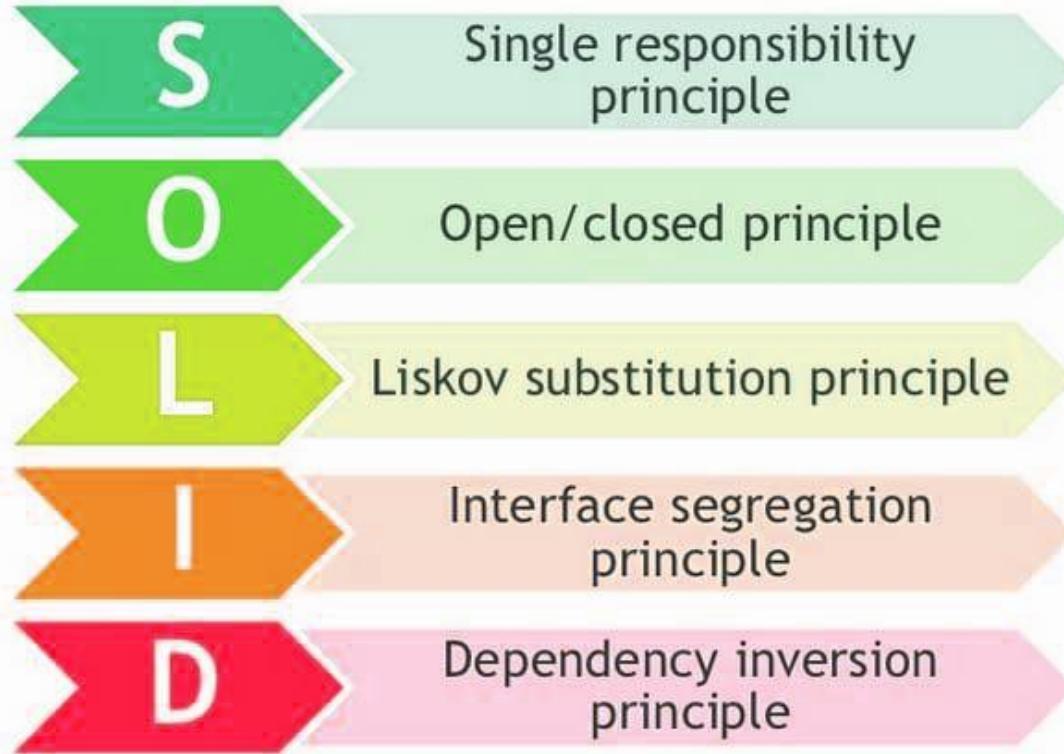
Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

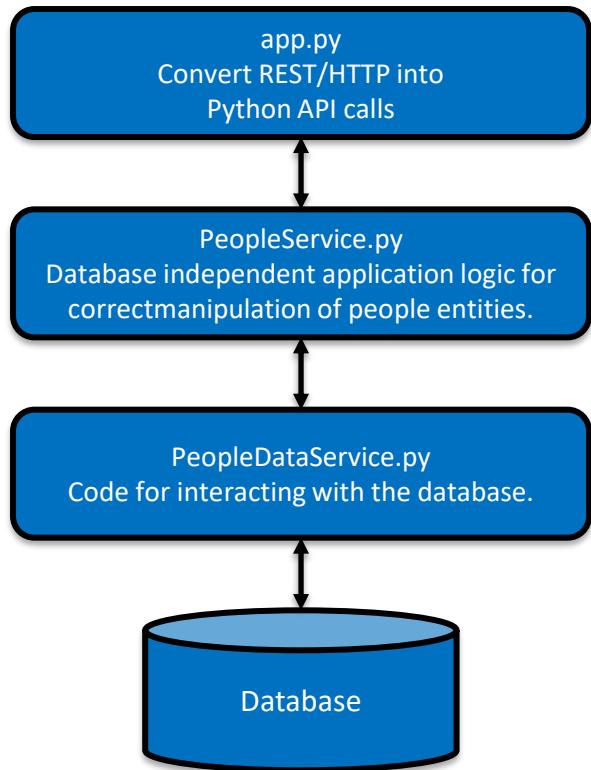
```
1 {  
2   "teamID": "BOS",  
3   "yearID": 2004,  
4   "stint": 1,  
5   "H": 200,  
6   "AB": 600,  
7   "HR": 100  
8 }
```

```
INSERT INTO  
batting(playerID, teamID, yearID, stint, H, AB, HR)  
VALUES ("willite01", "BOS", 2004, 1, 200, 600, 100)
```

# SOLID (SW) Design Principle



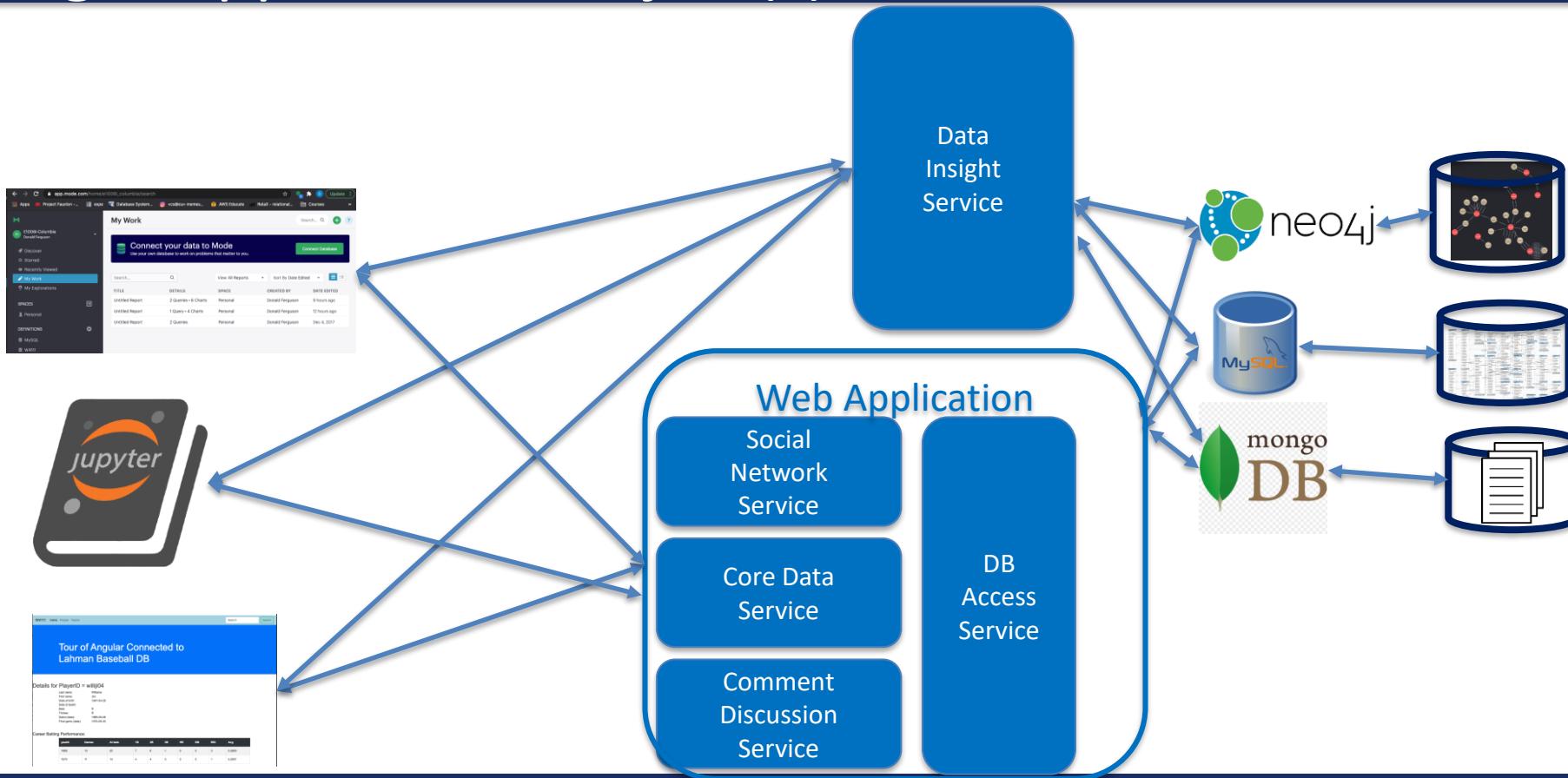
## Single Responsibility



# *HW3 (Project)*

## *To Be Refined and Simplified!*

# Target Application/Project(s) – Reminder



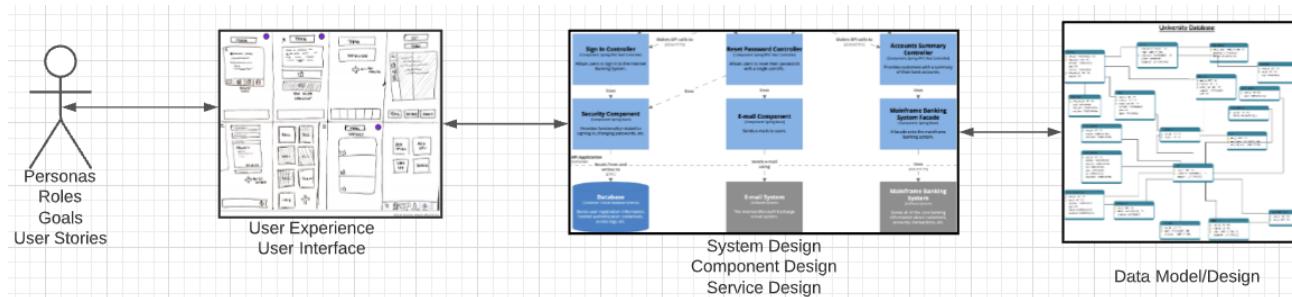
# Target Application/Project – Reminder (Lecture 1)

- That diagram was pretty confusing.
- Basically, what it comes down to is that there will be major subsystems:
  1. Interactive web application for viewing, navigating and updating data.  
The updates have to preserve *semantic constraints* and correctness.
  2. A decision support warehouse/lake that allows us to explore data and get insights.
- Programming and non-programming tracks will get experience with both, but
  - Non-programming track focuses on data engineering needed to produce (2).
  - Programming track will focus on (1).
- We will use *IMDB* and *Game of Thrones* because:
  - It has aspects and tasks interesting to both tracks.
  - We have an existing data set that we have been using.
  - There are interesting additional sources of data and use cases.

# *Operational System (Web Application)*

# Problem Statement – Modified (Lecture 1 Reminder)

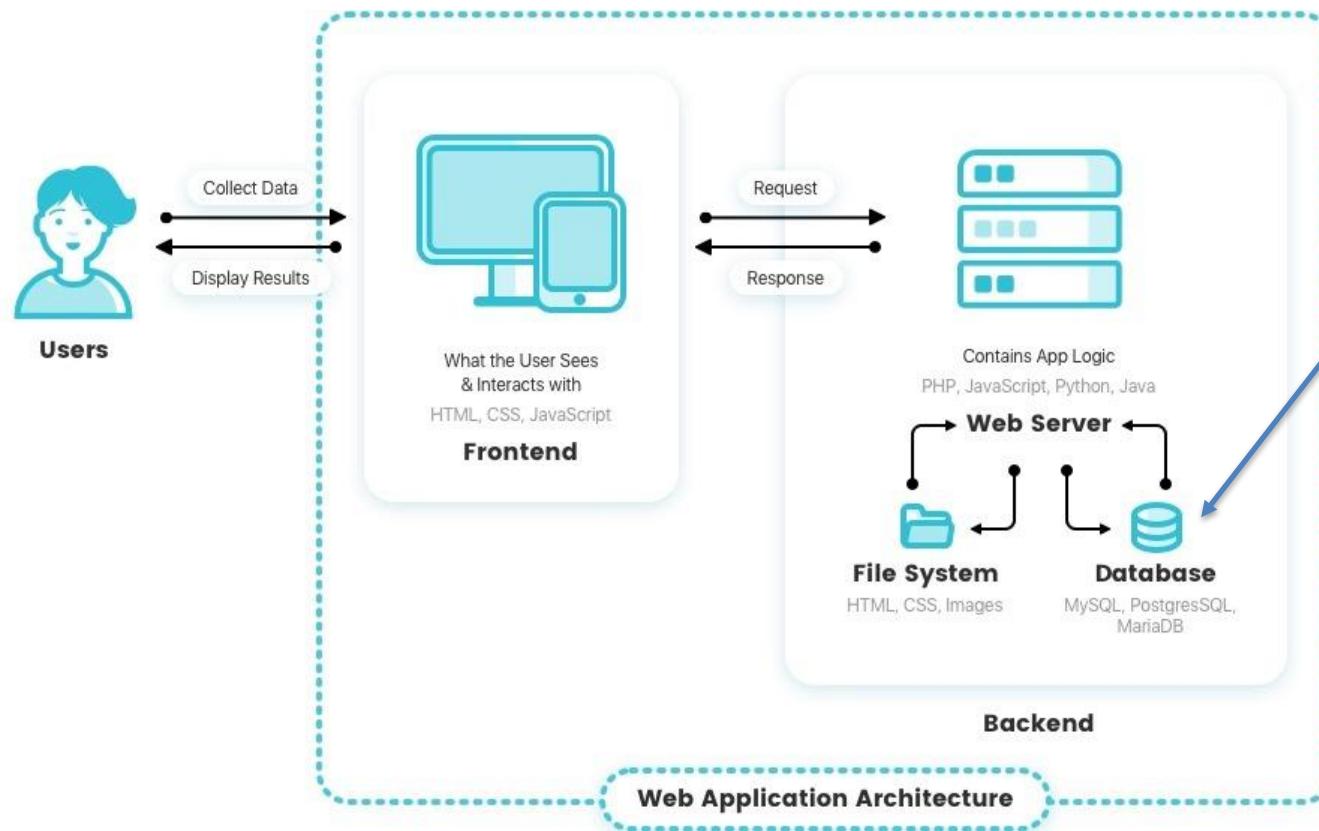
- We must build a system that supports ... ... operations in a ... ...
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



- The processes are iterative, with continuous extension and details.
- We will start implementing various *user stories*. Implementation requires:
  - Web UI
  - Paths
  - Data model and operations.

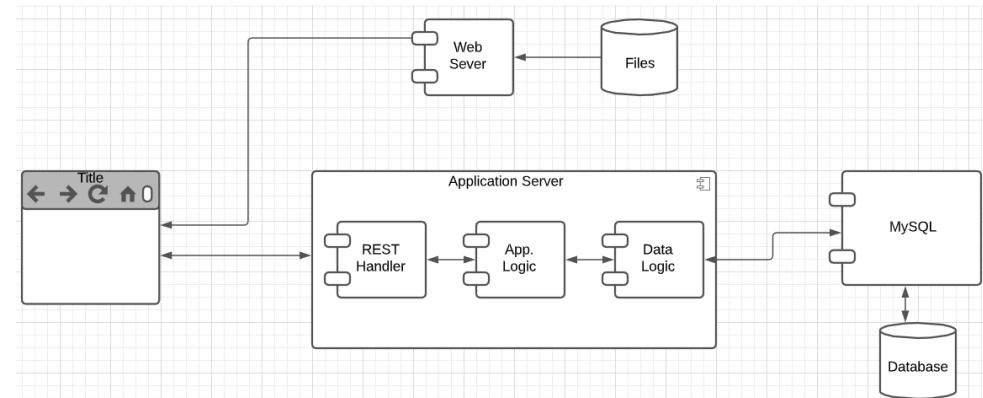
- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

# Web Application – Operational System



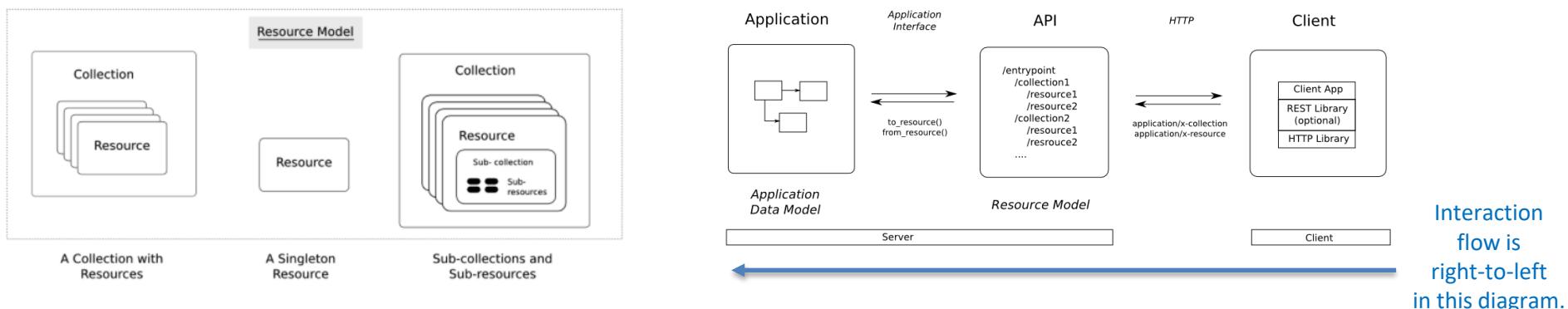
# User Story

- “In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system.”  
([https://en.wikipedia.org/wiki/User\\_story](https://en.wikipedia.org/wiki/User_story))
- Example user stories that I need to implement for the operational system:
  - “As a fantasy team manager, I want to search for players based on career stats.”
  - “As a fantasy team manager, I want to add a player to my fantasy team.
  - etc.
- I need to implement:
  - UI
  - Application logic
  - Database tables.



# Simple Example – A Resource

<https://restful-api-design.readthedocs.io/en/latest/resources.html>



- Resources: (<https://restful-api-design.readthedocs.io/en/latest/resources.html>)
  - The fundamental concept in any RESTful API is the resource. A resource is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it.
  - ... .... information that describes available resources types, their behavior, and their relationships the resource model of an API. The resource model can be viewed as the RESTful mapping of the application data model.
- APIs:
  - ... .... APIs expose functionality of an application or service that exists independently of the API. (DFF comment – the data)
  - Understanding enough of the important details of the application for which an API is to be created, so that an informed decision can be made as to what functionality needs to be exposed ... ....
  - Modeling this functionality in an API that addresses all use cases that come up in the real world ... ....

# CRUD ([https://en.wikipedia.org/wiki/Create,\\_read,\\_update\\_and\\_delete](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete))

- Definitions:
  - “In computer programming, create, read, update, and delete[1] (CRUD) are the four basic functions of persistent storage.”
  - “The acronym CRUD refers to all of the major functions that are implemented in relational database applications. Each letter in the acronym can map to a standard Structured Query Language (SQL) statement, Hypertext Transfer Protocol (HTTP) method (this is typically used to build RESTful APIs[5]) or Data Distribution Service (DDS) operation.”

CRUD	SQL	HTTP	DDS
create	INSERT	PUT	write
read	SELECT	GET	read
update	UPDATE	PUT	write
delete	DELETE	DELETE	dispose

- Do not worry about Data Distribution Service.
- For our purposes HTTP – REST
- Entity Set:
  - Table in SQL
  - Collection Resource in REST.
- Entity:
  - Row in SQL
  - Resource in REST

# REST API Definition

W4111 Fantasy Baseball API

1.0.0

OAS3

This is a simple API

Contact the developer

Apache 2.0

Servers  
https://virtserver.swaggerhub.com/donff/W4111FantasyBas... ▾

SwaggerHub API Auto Mocking

The screenshot shows the SwaggerHub interface for the W4111 Fantasy Baseball API. At the top, there's a navigation bar with tabs for 'Servers' (selected), 'Contact the developer', and 'Apache 2.0'. Below the servers section, there's a dropdown menu showing the URL: https://virtserver.swaggerhub.com/donff/W4111FantasyBas... ▾. The main content area is titled 'SwaggerHub API Auto Mocking'. It lists several resource groups: 'admins' (Secured Admin-only calls), 'developers' (Operations available to regular developers), and 'Fantasy Baseball'. The 'Fantasy Baseball' group is expanded, showing two operations: 'GET /fantasy\_baseball/teams' and 'POST /fantasy\_baseball/teams'. The 'POST' method is highlighted with a red border. Below this, there's a section for 'Real World' and a 'Schemas' section containing 'Team' and 'Player' definitions.

er HTTP in a similar way to web browsers and  
n a fundamental requirement of software

uiding constraints which must be satisfied if an

## Open API Definition

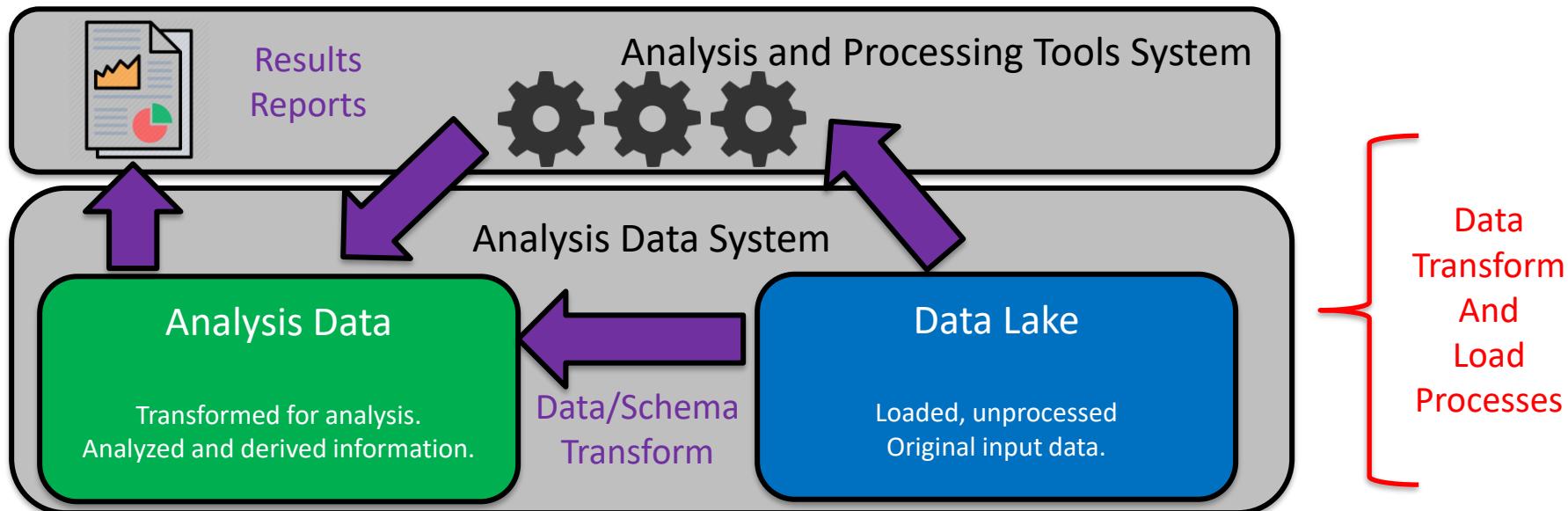
- API Tags/Groupings
- A resource has
  - Paths
  - Methods
- Schema (Data Formats)
  - Sent on POST and PUT
  - Returned on GET

This material is just FYI and to help with  
understanding concepts, mapping to DB, ...

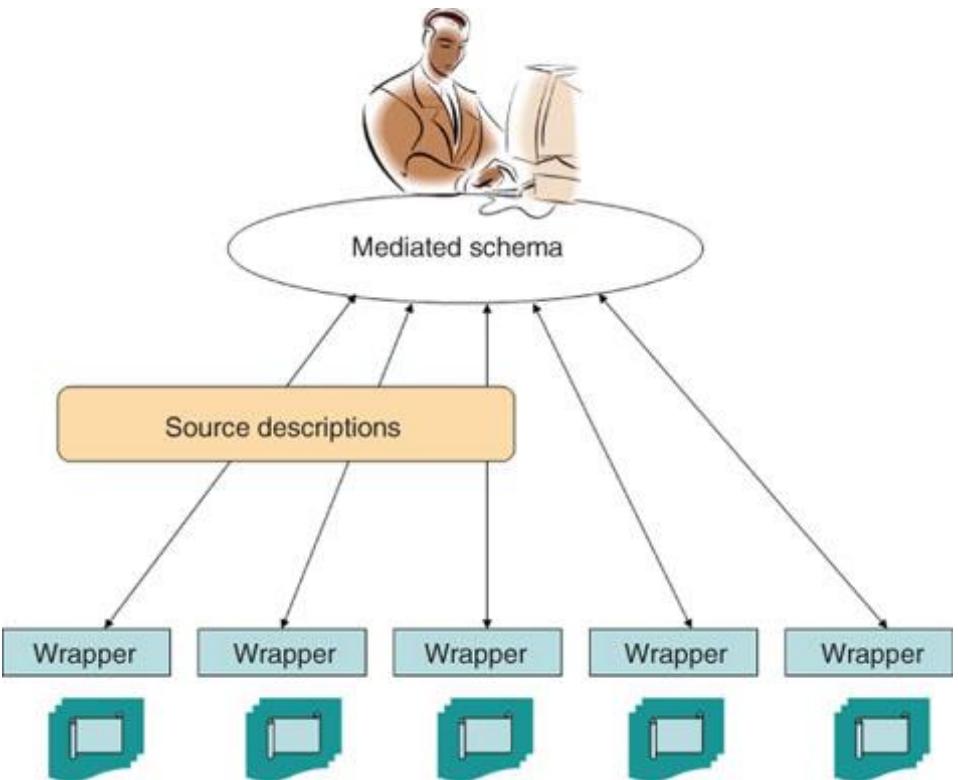
# *Analysis System*

# Analysis System – Fantasy Baseball Concept

- Focus is on the Analysis Data System (Primary Focus):
  - Data Lake is source data, imported and added to common database/model. (e.g. Lahman Baseball DB)
  - Analysis data is transformed data suitable for analysis, and analysis results. (e.g. Transformed Lahman's Data)
- Various analysis and processing tools use the data for insight, visualization, etc. (e.g. Jupyter, Pandas)



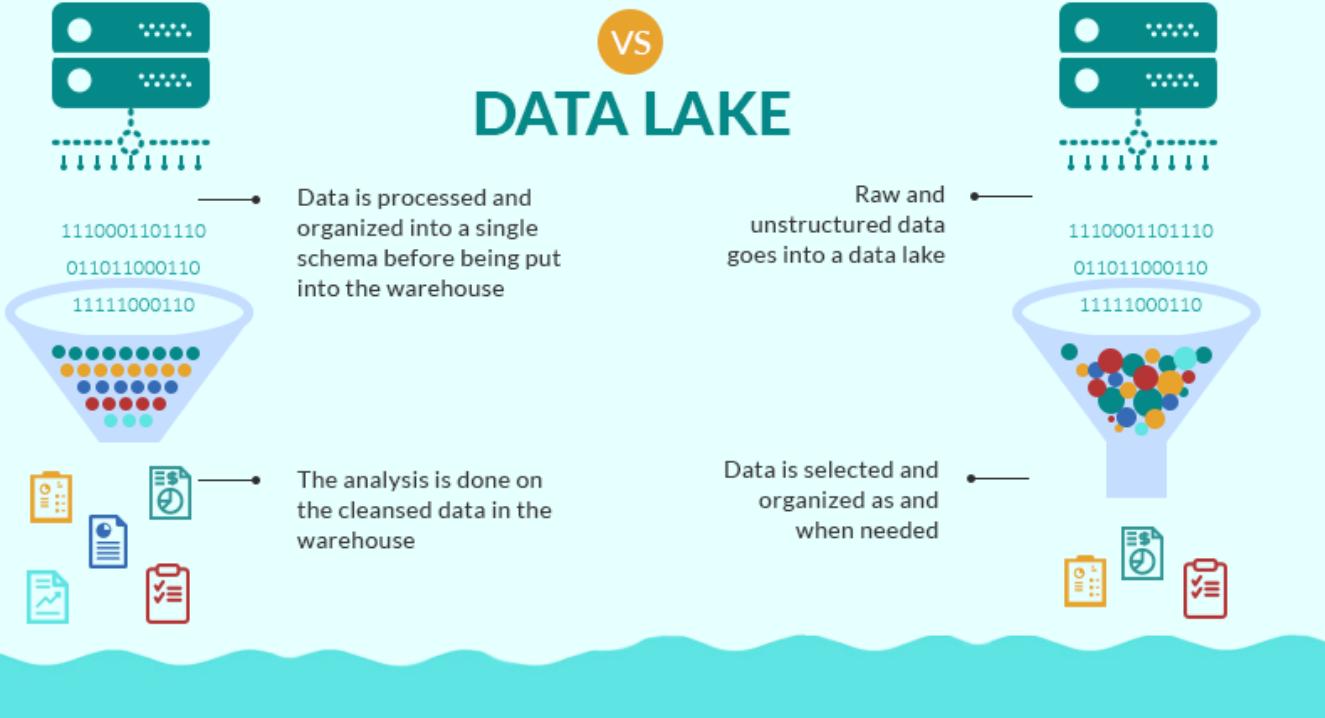
# Enterprise Information Integration



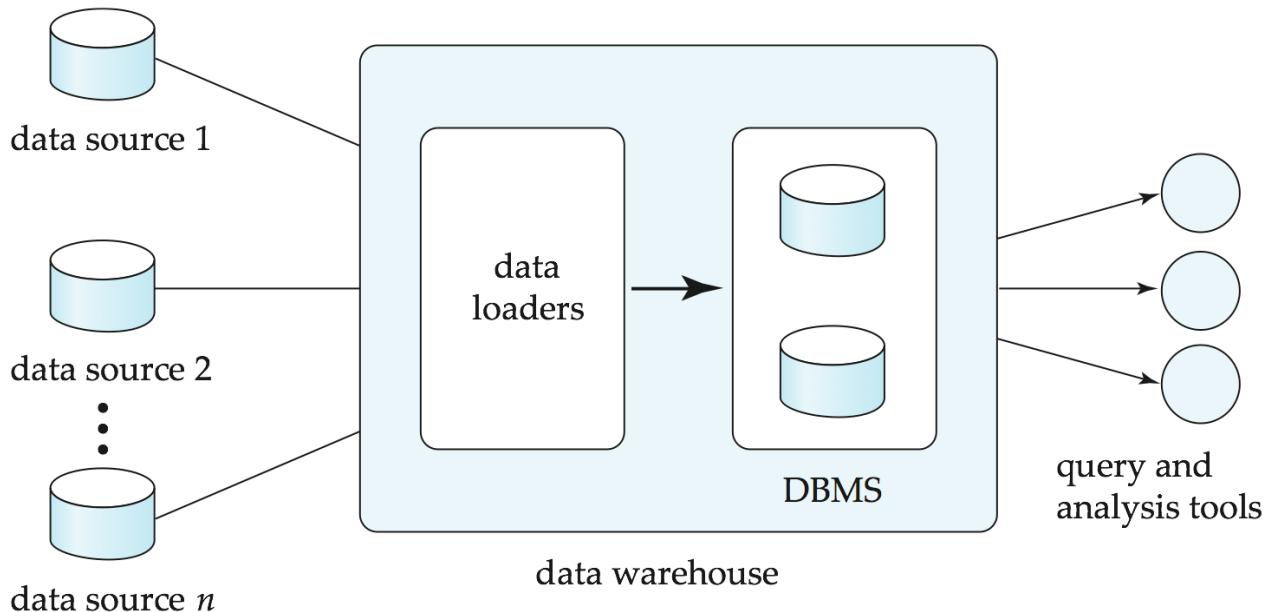
# Data Warehouse and Data Lake

## DATA WAREHOUSE

## VS DATA LAKE



# Data Warehousing

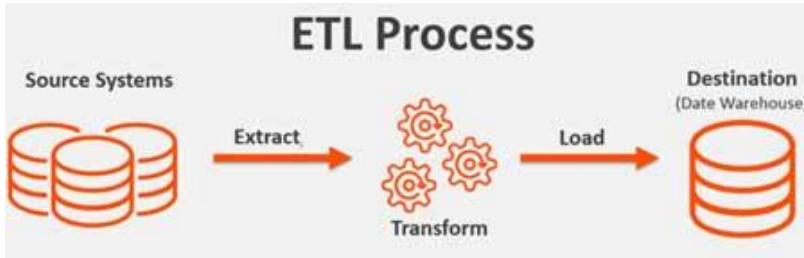


# Overview (Cont.)

- Common steps in data analytics
  - Gather data from multiple sources into one location
    - Data warehouses also integrated data into common schema
    - Data often needs to be **extracted** from source formats, **transformed** to common schema, and **loaded** into the data warehouse
      - Can be done as **ETL (extract-transform-load)**, or **ELT (extract-load-transform)**
  - Generate aggregates and reports summarizing data
    - Dashboards showing graphical charts/reports
    - **Online analytical processing (OLAP) systems** allow interactive querying
    - Statistical analysis using tools such as R/SAS/SPSS
      - Including extensions for parallel processing of big data
  - Build **predictive models** and use the models for decision making

# ETL Concepts

<https://databricks.com/glossary/extract-transform-load>



## Extract

The first step of this process is extracting data from the target sources that could include an ERP, CRM, Streaming sources, and other enterprise systems as well as data from third-party sources. There are different ways to perform the extraction: **Three Data Extraction methods:**

1. Partial Extraction – The easiest way to obtain the data is if the source system notifies you when a record has been changed
2. Partial Extraction- with update notification – Not all systems can provide a notification in case an update has taken place; however, they can point those records that have been changed and provide an extract of such records.
3. Full extract – There are certain systems that cannot identify which data has been changed at all. In this case, a full extract is the only possibility to extract the data out of the system. This method requires having a copy of the last extract in the same format so you can identify the changes that have been made.

## Transform

Next, the transform function converts the raw data that has been extracted from the source server. As it cannot be used in its original form in this stage it gets cleansed, mapped and transformed, often to a specific data schema, so it will meet operational needs. This process entails several transformation types that ensure the quality and integrity of data; below are the most common as well as advanced transformation types that prepare data for analysis:

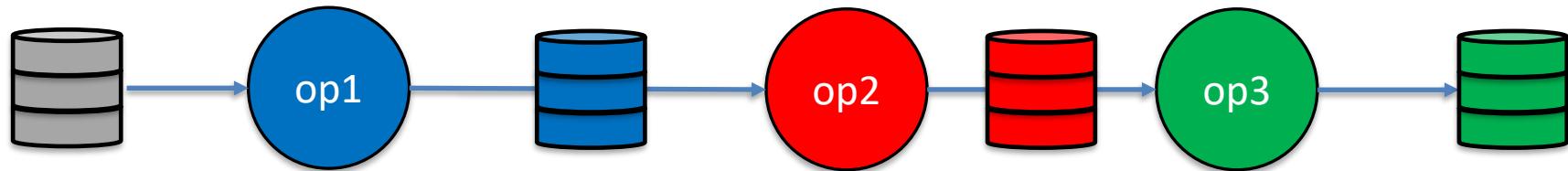
- Basic transformations:
- Cleaning
- Format revision
- Data threshold validation checks
- Restructuring
- Deduplication
- Advanced transformations:
- Filtering
- Merging
- Splitting
- Derivation
- Summarization
- Integration
- Aggregation
- Complex data validation

## Load

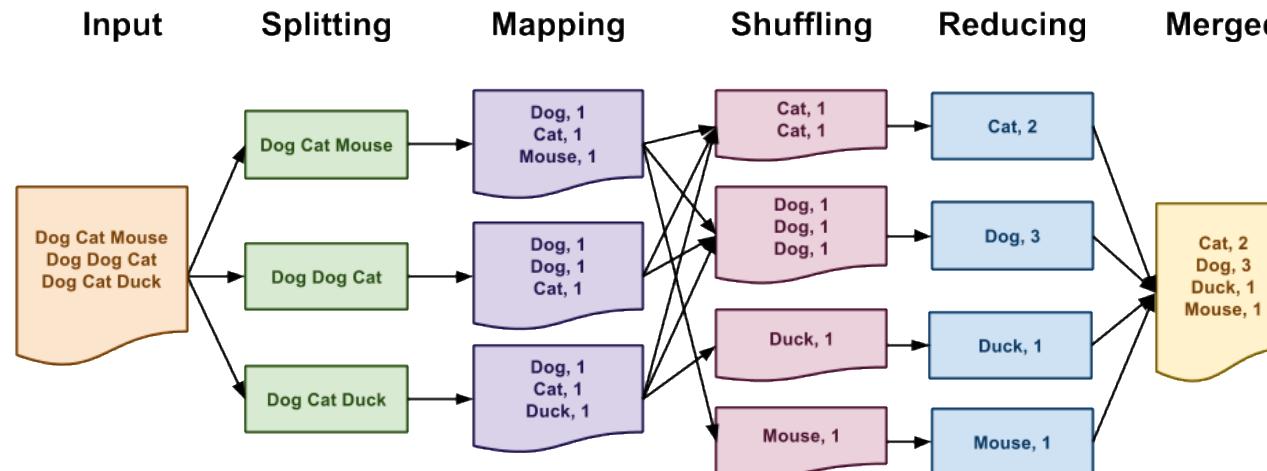
Finally, the load function is the process of writing converted data from a staging area to a target database, which may or may not have previously existed. Depending on the requirements of the application, this process may be either quite simple or intricate.

# MapReduce

MapReduce is a data flow program with relatively simple operators on the data set.



With each operator implemented in parallel on multiple nodes for performance.

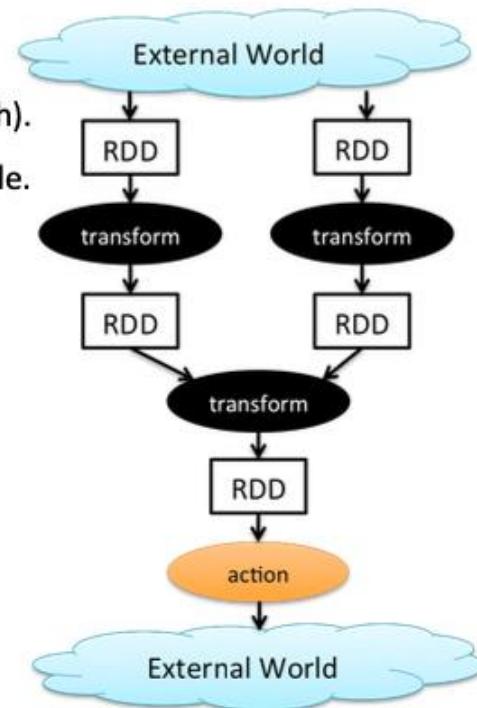
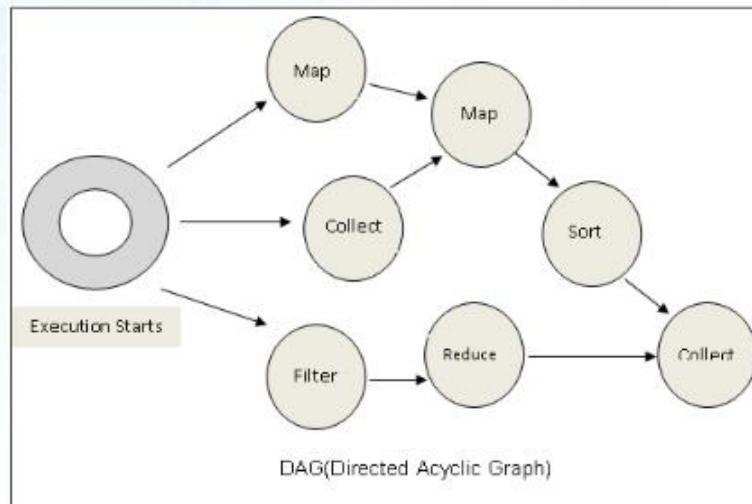


What is we want more complex “operators?”

# Algebraic Operations

- Current generation execution engines
  - natively support algebraic operations such as joins, aggregation, etc. natively.
  - Allow users to create their **own algebraic operators**
  - Support trees of algebraic operators that can be executed on **multiple nodes in parallel**
- E.g. Apache Tez, Spark
  - Tez provides low level API; Hive on Tez compiles SQL to Tez
  - Spark provides more user-friendly API

- All jobs in spark comprise a series of operators and run on a set of data.
- All the operators in a job are used to construct a DAG (Directed Acyclic Graph).
- The DAG is optimized by rearranging and combining operators where possible.



# Mapping to HW3a

- Installing, setting up, configuring, ... the software can be very, very complex.
- We are going to understand the concepts using MongoDB Aggregation Pipeline, which is simpler and more restricted.
  - Not graphical/DAG
  - Parallelism unclear
- Explain and demonstrate the concept of parallelism.