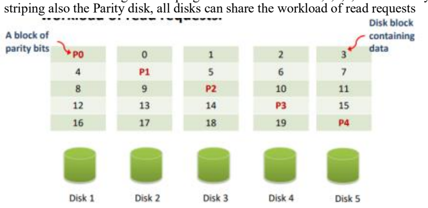
CH5 Functional dependency (FD) is a constraint between two sets of attributes in a relation from a database . It requires that the values of a certain set of attributes uniquely determine (imply) the values for another set of attributes. each X value is associated with precisely one Y value

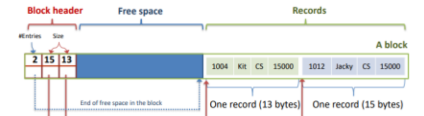
Armstrong's axioms 1. Reflexivity - if $\beta \subseteq \alpha$, then $\alpha \to \beta$. (BD->B)(AD->BD)
2. Transitivity - if $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$.
3. Augmentation - if $\alpha \to \beta$, then $\gamma\alpha \to \gamma\beta$.
4. Union - if $\alpha \to \beta$ and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$.
5. Decomposition - if $\alpha \to \beta\gamma$, then $\alpha \to \beta$ and $\alpha \to \gamma$.
6. Pseudo-transitivity - if $\alpha \to \beta$ and $\gamma\beta \to \delta$, then $\alpha\gamma \to \delta$

cannot derived a tighter FD from a looser FD. B->C then B->BC Attribute set closure (a^+) is the set of attributes that can be functionally determined by α eg F = {A → B, B → C} $\{A\}^+$ = { A, B, C } /{B} += { B, C }/ {C} += { C } /{A,B} += { A, B, C } The use of α Testing for superkey –α is a super key of R iff α + contains all attributes of R. Check if the decomposition of a relation is dependency preserving or not. Calculate FD closure F+ , which is an important tool in database normalization eg prove candidate key Prove that 1) C is a superkey. 2) Prove that C is minimum (no subset of C is a superkey). {C}+ = {A,B,C,D,E}. then is superkey Since C→A, C→AC (by Augmentation) FD closure F+ The set of ALL functional dependencies that can be logically implied by F is called the closure of F (or F+) Given a relation R(N, S, P) and the functional dependencies F = {N→S, N→P} find the FD closure F+ . Step 1. Treat every subset of R as α. Step 2. For every α, compute α + Step 3. Use α as LHS, and generate an FD for every subset of α + on RHS. Chapter 5B. Database Normalization The process of organizing the columns and tables of a relational database to minimize redundancy and dependency. 1. Lossless-join – Avoid the decomposition result in information loss. R1 ⋈ R2 ≠ R Testing Schema of R1 ∩ schema of R2 → schema of R1 OR Schema of R1 ∩ schema of R2 → schema of R2 R(A,B,C), F={B→C} Step 1 Find common attributes of R1 and R2 Step2: B → R1 (i.e., B → AB?) or B → R2 (i.e., B → BC?) either one is correct then is enough 2. Dependency preserving – Avoid the need to join the decomposed relations to check the functional dependencies when new tuples are inserted into the database. Consider R(A,B,C,D), F = {A → B, B →CD} :: F+ = {A → B, B →CD, A →CD, trivial FDs} If R is decomposed to R1 (A,B), R2 (B,C,D): F1 = {A → B, trivials}, the projection of F+ on R1, F2 = {B → CD, trivials}, the projection of F+ on R2 dependency preserving decomposition as: (F1 ∪ F2)+ = F+ (F = (F1 ∪ F2)3.Reduce redundancy – The decomposed relations Ri should be in Boyce-Codd Normal Form (BCNF). (There are also other normal forms like 3NF.) a relation R has no redundancy, or in Boyce-Codd Normal Form (BCNF), if the following is satisfied: For all FDs in F+ of the form α → β, where α ⊆ R and β ⊆ R, at least one of the following holds: α → β is trivial (i.e., β ⊆ α) //// α is a key (superkey) for R Testing check only the dependencies in the given F for violation of BCNF, rather than check all dependencies in F+ For example, given R(A,B,C); F = {A→B, B→C}, we only need to check if both {A}+ and {B}+ cover {A,B,C}. We do not need to derive F+ = {A→B, B→C, A→C, …etc} and check each FD because A→C already considered when computing {A}+ . However, if we decompose R into R1 and R2 , we have to use F+ instead. we must project F+ onto the relation (e.g., R2), not F eg F1 = {C→ABD, BD→AC }. 2. Determine if all FDs in F1 forms a key of R1 . 1. Is {C} a key in R1? Yes! Because {C}+={ABCD} = R1. 2. Is {BD} a key in R1? Yes! Because {BD}+={ABCD} = R1. 3. Since all non-trivial FDs in F1 forms a key, R1 is in BCNF. Chapter 6.File &Storage Cache memory is extremely fast memory that is built into a computer's central processing unit (CPU.(volatile storage) Its use is managed by the computer system hardware. We shall not be concerned about managing CPU cache storage in the database system. However, it is worth noting that database implementers do pay attention to cache effects when designing query processing data structure and algorithms. Main Memory Volatile storage. Fast access (in nanoseconds : 10–9 seconds). Generally too small (or too expensive) to store the entire database (of an enterprise). Capacities of a few Gigabytes. Capacities have gone up and cost-per-byte has decreased steadily and rapidly Magnetic Disk Primary medium for the long-term storage of data; typically stores the entire database. Access time: much slower than main memory. Non-volatile storage. Data are loaded into memory (a buffer) before accessed by DBMS. Read-Write Heads. Positioned very closely to the platter surface. Reads or writes magnetically encoded information. A disk has many platters. (碟) Each platter has two surfaces covered with magnetic materials, information is recorded on the surfaces. Each platter is divided into circular tracks . There are about 50,000 to 100,000 tracks per platter. (very dense) Each track is further divided into sectors. A sector is the smallest unit of data that can be read/written. Sector size is typically 512 bytes. Typical sectors per track: 500 -1000 (inner tracks) 1000 -2000 (outer tracks). To read/write data 1. [Seek] Position the head on the right track by moving the disk arms. 2. [Rotation] Spin the disk so that the start of data is under the head. 3. [Transfer data] Continue spinning and transfer the data. Access Time – the time between the request and the start of data transfer. This consists of: Seek time The time required to reposition the arm over the correct track. Around 2 to 30 milliseconds on typical disks, depend on the physical location of the data. Rotational Latency The time required to rotate the platter until the required sector is under the disk head. Around 4 to 11 milliseconds per rotation (15,000 revolutions per minute (rpm) to 5,400 rpm Data Transfer Rate – the rate at which data is retrieved from or stored to disks: Typical value : 25 to 300 megabytes per second (MBps). As multiple disks may share the same controller, we have to be aware of the controller's processing speed. Data block Data must be first transferred to main memory (buffer) before the DBMS can operate on them. The data transfer unit between disk and memory is called a Data block. Usually with size 4KB to 16KB (spans multiple sectors). When a single item is needed (e.g., an attribute value of a specific tuple), the whole block that contains the item is transferred. Reading / writing of a disk block is called an I/O operation. The time required to read/write a block depends on the block's location on disk Time for one I/O operation = seek time + rotational delay + transfer time Efficiency issue: Time to move data from/to disk usually dominates the cost of processing a query (CPU actions are in nano-seconds, and a block access is in milli-seconds!) Magnetic Tape Used primarily for offline backup (to recover from disk failures) and archival purpose. Non-volatile storage. Cost: very low. Access Speed: slow, and only sequential access. Optical Non-volatile storage. Access Speed: Read – much slower than magnetic disks (especially on seeks, i.e., random access); Write – even more slower than magnetic disks. Capacity: DVD (4.7GB to 17GB) BDXL (100GB to 128GB) Usually write-once, read many (WORM) optical disks are used for archival storage. Flash Memory Non-volatile storage. Access Speed: Read/write : in "page" granularity, corresponds to disk "sector" (typically 4 KiB). Erase: A page can only be written to, after it is erased. With high latency, typically in ms time. 20 Each cell has limited program/erase lifetime (thousands, for modern devices) – Cells become slowly less reliable. Storage Hierarchy Put all data on disks Data must be maintained after power failure. Storage capacity must be big enough for all data. Use memory for temporary storage and manipulation of data during queries Selected data are transferred to memory for fast processing. Updates are first performed in memory and later written back to disk. Backup data on tertiary storage Periodically backup the contents of DB on tapes. Reliability & Efficiency Reliability - Hard disks may fail, but we don't want to lost our data. Efficiency - Disks are slow compare with the speed of CPU. Solutions Mirroring. Data striping. Error-correction schemes Disk Failure MTTF (Mean time to failure) – average time the disk is expect to run continuously without failure. Mean time to data loss depends on MTTF and MTTR

how disks are organized. Mirroring Storing a redundant copy of data in another disk(s). Mean time to data loss will be much longer Efficiency The rate at which read requests can be handled is doubled - read requests can be set to all disks. Note: The speed of each read (or query) is the same as in a single-disk system. Data Striping Data are partitioned to several disks (e.g., first block to disk 1, second block to disk 2, etc.) Faster read can be achieved by parallel read Parity check with parity disk Although the data on Disk 3 is lost, we can reconstruct the data in Disk 3 based on the data on Disk 1,2,4 and P RAID Redundant Arrays of Independent Disks. Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rate, but does not improve reliability. RAID has various levels – with different combinations of mirroring, striping and error-correction strategies. RAID 0 – Striping only, no mirroring. RAID 1/ RAID 10/ RAID 1+0 – Mirroring with striping. Others: RAID level 2,3,4,5,6. RAID 5 By striping also the Parity disk, all disks can share the workload of read requests



File Organization A database store the records in file(s). Many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself Each file is logically partitioned into fixed-length storage units called blocks, which are the units of both storage and data transfer. A block may contain several records. Assumption : No record is larger than a block. Fixed-length records The length of every record is fixed. Record access is simple, but records may cross blocks. Modification: Do not allow records to cross blocks, let those areas as unused area Free list Store the address of the first deleted record in the file header. Use the first record to store the address of the second deleted record, and so on. Variable-length records Variable-length records arise in database systems in several ways: Storage of multiple records types in the same block (e.g, Some tuples of the Instructor table, and some tuples of the Department table stored together in one block). Record types that allow variable lengths for one or more fields. (e.g., VARCHAR(250), TEXT …etc) Slotted-page structure is commonly used for organizing variable-length records within a block.



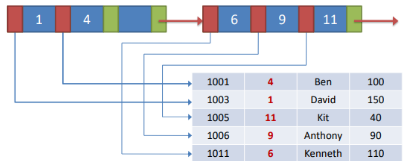
Organizing records in files Heap file No ordering of records, can place anywhere Adv: Simplicity – stores every record in any empty space in any blocks. Div: New blocks are allocated or destroyed dynamically; i.e., blocks in a file may be scattered over the disk. Sequential file Store records in sequential order, based on the value of the search key of each record. Sequential file is designed for efficient processing of records in sorted order based on some search key. Has to maintain the order during record insertion. Locate the record in the file that comes before the record to be inserted in search-key order. If there is a free slot (maybe after previous deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. Hashing A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed. Multitable clustering We may put ≥ 2 related relations in the same file, to achieve faster joins comparison Heap file is the cheapest to maintain. But have to scan all data to locate a specific record. Sequential file helps query evaluation. But it is difficult to maintain a sequential file. Clustering file helps joins and finding related records over different relations. Accessing data on only one relation may suffer. Variable size records may be difficult to handle Buffer Buffer manger DBMS seeks to minimize the number of blocks transfers between disks and memory. Buffer - Portion of memory available to store copies of disk blocks. Buffer Manager – A program responsible for allocating buffer space in main memory and moving blocks between disk and memory (so that disk I/O is minimized). Buffer miss Step 1. Read data in Block 1 Step 2. Block 1 is not in the buffer, so it is retrieved from disk (1 Disk I/O). Step 3. The required data in block 1 is sent to the application. Buffer hit Step 1. Read data in Block 2. Step 2. The required data is in the buffer, return to users without disk access (No I/O). Write operation Step 1. Update record in block 2 Updates are done in memory only. The result will be reflected on disk when the buffer is flushed back to disk (or under other reliability requirements). Buffer full Step 1. Read data in block 5. Since block 5 is not in buffer, we need to fetch it from disk. But no space in buffer space Step 2. Suppose we free the buffer space that was used by block 7, simply overwrites the buffer slot. If we free the buffer space that was used by block 2, we need an extra step to write the updated data of block 2 from memory to disk first. Buffer replacement policy Most operating systems replace the block that was the least recently used – LRU strategy The intuition behind LRU If a block is not used for a long time, it is not likely that it will be accessed again very soon. This uses the past patterns of block accesses as a predictor of future accesses LFU – Least frequently used., etc. Data dictionary Data dictionary (also called system catalog) stores metadata, i.e., data about data. e.g., Information about relations (names of relations, names & types of attributes, integrity constraints, views) Statistical data (e.g., number of tuples in each relation). Physical file organization (sequential, hashing, etc.) Frequently accessed by the buffer manager and query optimizer and therefore stays in the memory for fast access. Tour What's the maximum seek time? The time it takes the head to move n tracks is 1 + 0.001n millisecond There are 10,000 tracks per surface, so the max seek time is 1 + 0.001 * 10,000 milliseconds = 11 milliseconds – Precisely, 1 + 0.001 * 9,999 = 10.99 milliseconds because it requires move across 9,999 tracks to travel from the outmost sector to the inner most one. (3 ms,us, ns,ps) What's the maximum rotational delay? Max time is when we need a full revolution – 10,000 rpm → 1/10,000 minute = 6 milliseconds What's the transfer time of an 8,192-byte block on the track containing 1,000 sectors? There are (on average) 1,000 sectors of 512 bytes per track 8,192 / 512 = 16 sectors, covering 360 * (16 / 1,000) = 5.76 degrees – Transfer time: (16 / 1,000) * 6 = 0.096 milliseconds How many cylinders does the disk have? – the number of cylinders is the number of tracks per surface - 10,000 cylinder

- A file has r = 20,000 student records of fixed length. Each record has the following fields:
  - name (30 bytes), ssn (9 bytes), address (40 bytes), phone (9 bytes),
  - dob(8 bytes), gender(1 byte), major (4 bytes), minor (4 bytes),
  - classcode (4 bytes), and degreeprogram (3 bytes).

An additional byte is used as a deletion marker. The file is stored on a disk whose block size is B = 512 bytes.

- **Question:** Calculate the record size R in bytes.
  - R = 30 + 9 + 40 + 9 + 8 + 1 + 4 + 4 + 4 + 3 + 1 = **113 bytes.**
- **Question:** How many records fit in a disk block, assuming an unspanned organization?
  - recordPerBlock = floor(B/R) = floor(512 / 113) = **4 records.**
- **Question:** How many blocks will the file occupy?
  - blocksOccupied = ceiling(r / recordPerBlock) = ceiling(20,000 / 4) = **5,000 blocks.**

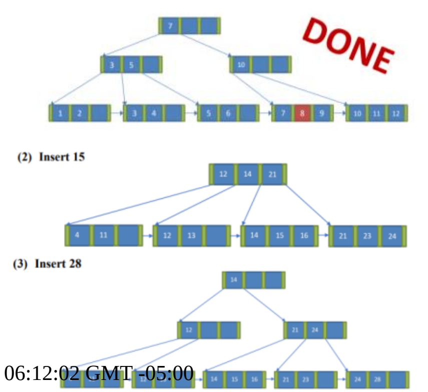
: Is 256 bytes a valid block size? How about 2,000 bytes or 51,200 bytes? – A block consists of multiple consecutive sectors – 256 bytes and 2,000 bytes are invalid as they are not a multiple of 512bytes. – 51,200 bytes is valid (1 block = 100 sectors) If the file is stored sequentially on disk, with record 1 on block 1 of track 1, what is the first record stored on block 1 of track 1 on the next disk surface? – 1 block = 1,024 bytes = 2 sectors, containing 10 records – 1 track can hold 50 sectors or 25 blocks or 250 records – The 1st record on block 1 of track 1 on the next surface is record number 251. Chapter 7. Indexing Index is used to speed up access to desired data. Search key An attribute or a set of attributes used to look up records in a file Primary index - An index whose search key also defines the sequential order of the file. E.g., Access staff records through staffID 1. Ordered Indices – Search keys are sorted in the index Example: indexed-sequential file, B + -tree. 2. Hash Indices – Search keys are distributed over different buckets using a hash function. Index evaluation factors Each indexing technique must be evaluated on the basis of these factors Access types – The types of access that are supported efficiently (e.g., equality search or range search? Single attribute search or multi-attribute search? Access time – The time it takes to find a particular data item, or a set of items. Insertion / deletion time / Space overhead Properties of B^+ -tree All paths from root to leaf are of the same length Balanced Binary Search tree minimizes the number of key comparisons for finding a search key. Why don't we use balanced binary search tree in Database? Because we want to minimize the number of block retrieval in answering a query (i.e., number of tree nodes to be accessed) rather than the number of key comparisons We need a tree which is Node size = 1 block (A node can contain more than one keys) Low in height Balanced A node contains up to n-1 search-key values, and n pointers. The search-key values within a node are kept in sorted order. 1. Leaf node A leaf node has at least ⌈(n – 1)/2⌉ and at most (n – 1) values, where n is the number of pointers. The last pointer is used to chain together the leaf nodes in search-key order. The pointer before a search-key value points to the record that contains the search-key (assume no duplicate values).



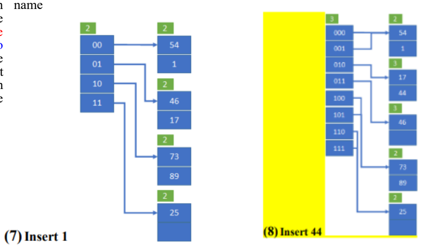
1001	4	Ben	100
1003	1	David	150
1005	11	Kit	40
1006	9	Anthony	90
1011	6	Kenneth	110

2. Non-leaf node Non-leaf nodes must hold at least ⌈n/2⌉, and at most n pointers. E.g., with n = 4, a non-leaf node contains at least 2 pointers, and at most 4 pointers. E.g., with n = 4, a non-leaf node contains at least 2 pointers, and at most 4 pointers. The pointer on the left of a key K points to the part of the subtree that contains those key values less than K. The pointer on the right of a key K points to the part of the subtree that contains those key values larger than or equal to K. Searching Step 1. Traverse from root to leaf. Step 2. Search in the leaf node. Step 3. Follow the pointer in the leaf node to retrieve the record. Insertion Assume no duplicate entries are inserted, insertion is simply searching + insert entry. If a leaf node is full, node splitting has to be performed. 1 2 3 5 Step 1. Create one more node and distribute the first ⌈n/2⌉ records to one node and the remaining to the other node. Step 2. Parent nodes (non-leaf nodes) have to be updated accordingly. Deletion Find the record to be deleted. Remove it from the file and from the leaf node (if present) If the leaf node has too few entries due to the removal: Try to MERGE the node with its sibling node. Try to REDISTRIBUTE the entries if MERGING fails.

- Suppose each B^+-tree node can hold up to 4 pointers and 3 keys. Insert **3, 4, 5, 7, 1, 2, 10, 12, 11, 6, 9,** and **8** in order into an initially empty B^+-tree. Show the tree after each insertion.



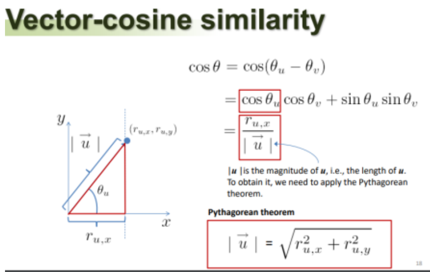
Assume that the B+ tree has a height of "H", and each non-leaf node can have a maximum of "M" children (keys per node). We can't find "H" and "M" without more information, so we'll leave them as variables for now. To answer the SQL query, we need to traverse the B+ tree from the root down to the leaf level and then retrieve the two matching records. The number of block accesses required for this operation would be: 1. Traversing the B+ tree: "H" block accesses (one for each level of the tree). 2. Reading the two matching records: 2 block accesses (assuming each matching record is in a different block). Therefore, the total number of disk block accesses is: H + 2 Without more information on the structure of the B+ tree index, we cannot provide a specific number for the block accesses. However, we have expressed the total block accesses as a function of the height"H"of the B+ tree. Hashing A kind of un-ordered indices. The entries are divided into buckets. A bucket is a unit of storage containing one or more records (a bucket is typically a disk block). No order within the index entries. In a hash file organization, we obtain the bucket of a record directly from its search-key value using a hash function. A hash function h() is a function that map the set of all search-key values to the set of all bucket addresses. Records with different search-key values may be mapped to the same bucket (Collision) Static hashing Hash function : K mod 4 Each bucket can hold 2 entries. Problems Database grows with time. If the initial number of buckets is too small, performance will degrade due to too much overflow. If we anticipate the database size and allocate space accordingly, a significant amount of space is wasted initially Solution Dynamic hashing (e.g., extendable hashing) Extendable hashing if full, need changing The bucket address table has a hash prefix value n. The first-n bits of the hash values are used to locate the address table entry for the pointer pointing to the bucket containing records with that hash value. Splitting a bucket If the hash prefix = i j Only 1 pointer in address table is pointing to bucket j. We increment hash prefix by 1, and double the size of the bucket address table. Reinsert the entries in the original bucket. If the hash prefix > i_j (i.e., the indicator in the bucket address table is larger than the indicator in bucket) Create a new bucket z, and set i j and i z to old i j + 1. There must be a number of address table entries pointing to bucket j originally, change the lower half of such entries so that they point to bucket z. Remove and re-insert each record in bucket j. Hash indexes v.s. B+ -tree Hash indexes have different characteristics than B+ -tree They are used only for equality comparisons that use the = operator (but they are very fast). 91 They are not used for comparison operators such as < that find a range of values. The optimizer cannot use a hash index to speed up ORDER BY operations. Defining index in SQL CREATE INDEX <index-name> ON <relation-name> (<attribute-list>) [index_type] ///// Use CREATE UNIQUE INDEX to indirectly specify and enforce the condition that the search-key is a superkey. To remove an index DROP INDEX <index-name

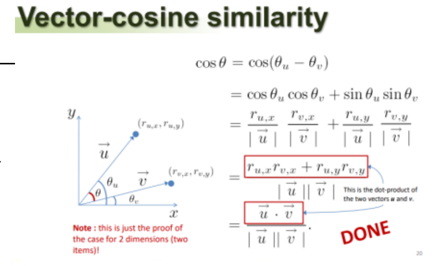


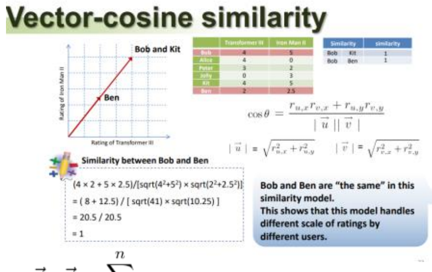
CH8 Collaborative Filtering Common insight: personal tastes are correlated Memory-based CF User-based /Item-based Model-based CF Clustering/ Bayesian belief networks/ Latent semantic analysis Memory-based CF Two core steps: Finding users who are similar to the active user. Produce a prediction for the active user based on the ratings of the "similar" users Measuring similarity There are various methods to represent the similarity of two users in memory-based CF. Vector Cosine-based similarity Correlation-based similarity Vector-cosine similarity User modeling: Each user is modeled as a "vector" of ratings. Similarity measure: The inner angle formed by the two users' vectors. Rationale - If the inner angle of the vectors of two users is smaller, then they are more similar (by scaling, range from 0 to 1) We obtain the cosine of the Θ angle. Then the value of 1 means the most similar, 0 means not similar.

$$\cos\theta = \cos(\theta_u - \theta_v)$$

d = sqrt[(ru,x -rv,x)^2 + (ru,y -rv,y)^2]
Distance = sqrt[(cos θu - cos θv)^2 + (sin θu - sin θv)^2]
= sqrt[cos θu^2 - 2 cos θu cos θv + cos θv ^{2} + sin θu ^{2} - 2 sin θu sin θv + sin θv ^{2}]
= sqrt[- 2 cos θu cos θv - 2 sin θu sin θv + 2] …(i)
d^2 = 1^2 + 1^2- 2(1)(1)cos θ
= 2 - 2 cos θ…….(ii)
Combining (i) and (ii)
2 - 2 cos θ = - 2 cos θu cos θv - 2 sin θu sin θv + 2
cos θ = cos θu cos θv + sin θu sin θv

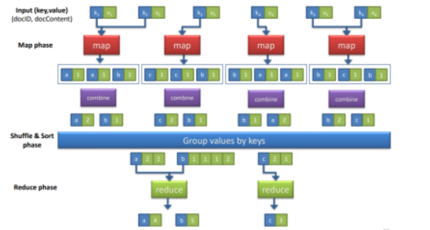






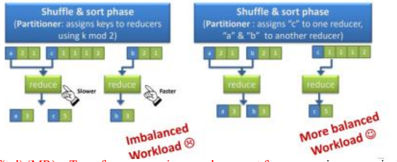
$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$$

CH9 MapReduce features By providing a data-parallel programming model, MapReduce can control job execution in useful ways. Automatic division of a MapReduce job into a number of map tasks and reduce tasks. Automatic placement of computation near data (data locality). Automatic load balancing (distribute the load evenly to the nodes). Automatic recovery from failures and stragglers. User focuses on application, not on complexities of distributed computing 1. Map phase A single master node controls job execution on multiple slaves nodes. Data locality – process local data to reduce transmitting data through network. Hadoop has a scheduler that starts tasks on the node that holds a particular block of data (i.e., on its local drive) needed by the task. Mappers save outputs to local disk. Allows recovery if a reducer crashes. 2. Shuffle and sort phase Intermediate key-value pairs must be grouped by key. Accomplish synchronization. Accomplished by a large distributed sort involving all the nodes that executed the map tasks and all the nodes that will execute the reduce tasks. The reduce computation cannot start until all mappers have finished emitting (key, value) pairs and all intermediate (key, value) pairs have been shuffled and sorted. 3. Reduce phase A reducer in MapReduce receives all values associated with the same key at once. Programmers can specify the number of reducers. Reduce method is called once per intermediate key. A reducer may execute more than one reduce method. E.g., Reducer A processes the intermediate keys "How", "many" and "happy" (3 reduce methods called) in our running example. Reducers also write their outputs to disk. Fault Tolerance If a Map/Reduce task crashes…Retry on another node, is it possible? OK for a map task because it has no dependencies. OK for a reduce task because map tasks output intermediate key-value pairs on disk. If a node crashes… Re-launch its current tasks on other nodes. Re-run any map tasks the crashed node previously ran because the outputs were lost along with the crashed node If a task is going slowly… Speculative execution - Launch second copy of task on another node. Take the output of whichever copy finishes first, and terminate the other copy. Very important strategy in large cluster Stragglers occur frequently due to failing hardware, bugs, etc. A single straggler may noticeably slow down the overall performance. E.g., A single straggler map task can delay the start of the shuffle and sort phase Inverted Indexing Inverted indexing: Nearly all search engines today rely on a data structure called an inverted index, which given a keyword provides access to the list of documents that contain the keyword. "inverted" because data usually data exist in "one document, many keywords" format. Inverted list is "one keyword, many document IDs" and therefore it is called "inverted". Why inverted index is useful? E.g., To find which document(s) contain the keywords "Apple Computer" , we simply join the two inverted lists to find the result (i.e., 3.html) Problem: Massive input data – E.g., The indexed web contains at least 3.93 billion web pages in 18 April 2023 Solution: Impossible to store the documents and build the inverted index using one computer. We need a cluster of machines to store the documents and compute the inverted index in parallel. MapReduce We only need to provide the Mapper function and the Reducer function if we use the MapReduce framework. What are the 1. Input/ Output key-value pairs and the 2. logic in the two functions? Mapper Input - <docID, docContent> pairs. Output – Emit <keyword, docID> as intermediate key value pairs, and only emit once for each keyword appear in the document docID. Reducer Input - <keyword, list(docID1 , docID2 , docID3 , …etc)> pairs. Output - <keyword, sorted list of docIDs> pairs Combiner – Allow mappers to perform local aggregation (after mapper) before shuffle sort phase Partitioner - Allow programmers to divide up intermediate key space and assigning (key,value) pairs to reducers Combiner Local aggregation – Combine some key-value pairs in each mapper to reduce the number of key-value pairs that pass into the shuffle and sort phase. E.g., In the wordCount example, instead of emitting <"how", 1> twice if a document has two "how"s, We emit <"how", 2>. (buffer space has to be allocated to store the intermediate word count)



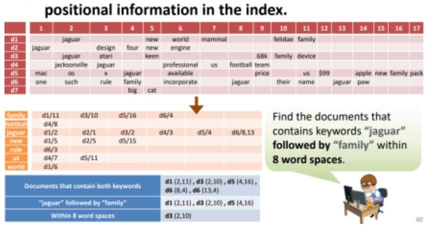
Partitioner The simplest partitioner involves computing the hash value of the key and then taking the mod of the value with the number of reducers. This assigns approximately the same number of keys to each reducer. (We want to balance the reducers' workload) Problem : Partitioner only considers the key and ignores the value. (word count example, Some words may occur more frequent than others, Therefore even we assign the same number of keys to each reducer, the reducers' workload may still be imbalanced.)

- The partitioner allows programmer to specify **which reducer will be responsible for processing a particular key**.

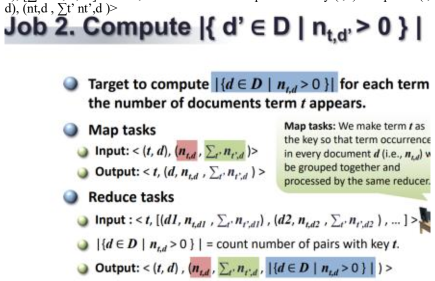


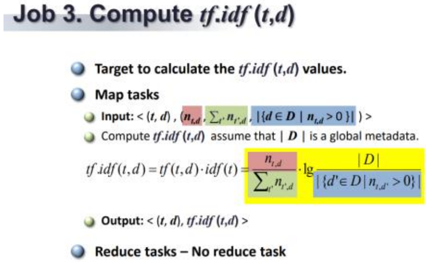
tf.idf(t,d) (MR) Term frequency – inverse document frequency , is a numerical statistic that is intended to reflect how important a term t (keyword) is to a document d in a collection or corpus. The tf.idf(t,d) value increases proportionally to the number of times a term t appears in the document d, but is offset by the frequency of the term t in the collection, which helps to adjust for the fact that some words appear more frequently in general may not be very informative to describe a document d. Preprocessing 1. Tokenization Separate text into tokens (words) Challenges: Compound words: hostname, host-name and host name. Break into two tokens or regroup them as one token? In any case, lexicon and linguistic analysis needed Preprocessing 2. Stemming Merge different forms of the same word, or of closely related words, into a single stem. Morphological stemming - Remove bound morphemes from words, such as remove final -s, -'s, -ed, -ing, -er, -est. Lexical stemming - Merge lexically related terms of various parts of speech, such as policy, politics, political or politician . Phonetic stemming - Merge phonetically related words: search despite spelling errors, such as happiness and happiness Preprocessing 3. Stop word removal Remove uninformative words from documents, in particular to lower the cost of storing the index Inverted index After all preprocessing, construction of an inverted index. Index of all terms, with the list of documents where this term occurs. Small scale: disk storage, with memory mapping (cf. mmap) techniques; secondary index for offset of each term in main index. Large scale: distributed on a cluster of machines; hashing assigns the machine to responsible for some term(s). Updating the index is costly, so only batch operations (not one-by-one addition of term occurrences).

- **Phrase queries, NEAR operator: need to keep positional information in the index.**



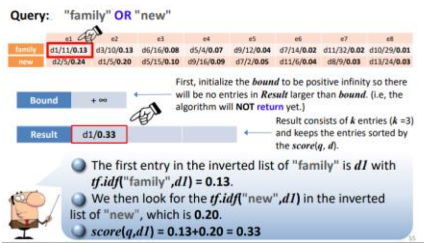
Term frequency tf (t,d) The term frequency tf (t,d) is the number of occurrences of a term t in a document d (denoted by $n_{t,d}$), divided by the total number of terms in d (denoted by $\sum_{t'} n_{t',d}$). E.g., tf("jaguar",d1) = 1/6 Inverse document frequency idf(t) How informative is a term to a document Insight: Terms occurring rarely in the document collection as a whole: more informative. The inverse document frequency idf(t) of a term t is obtained from the division of the total number of documents |D| by the number of documents where t occurs (denoted by $|\{d \in D \mid n_{t,d} > 0\}|$) . idf("jaguar") = lg (7/6) = 0.22 idf("jacksonville") = lg (7/1) = 2.8 The term "jacksonville" is more informative to identify a matched document (say d3) than "jaguar" because "jaguar" is more common, "jacksonville" is rarely happen (more specific). inverted indices The inverted lists are extended by adding tf.idf(t,d) for each entry, sorted in descending order of tf .idf(t,d). eg tf("family", d1) = 1/6 ; idf("family") = lg (7/4); tf.idf("family", d1) = 1/6 lg (7/4) = 0.13; we need $n_{t,d}$ the number of occurrences of term t in document d. $\sum_{t'} n_{t',d}$ the total number of terms in document d. $|\{d \in D \mid n_{t,d} > 0\}|$ the number of documents term t appears. | D | the total number of documents Job 1. Compute $n_{t,d}$ and $\sum_{t'} n_{t',d}$ Target to compute $n_{t,d}$ for each term t - the number of occurrences of term t in document d. Map tasks Input: <d, content of d> Output: < (t, d), 1> Reduce task Input: < (t, d), [$\sum_{t'} n_{t',d}$,…] > $n_{t,d}$ = count number of pairs with key (t,d). Output: < (t, d), ($n_{t,d}$, $\sum_{t'} n_{t',d}$)>



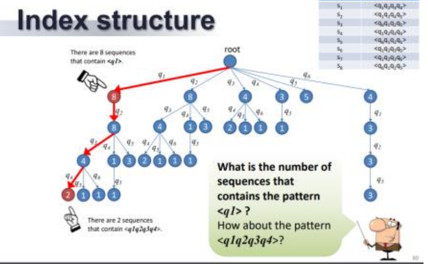


$$tf.idf(t,d) = tf(t,d) \cdot idf(t) = \frac{n_{t,d}}{\sum_{t'} n_{t',d}} \cdot \lg \frac{|D|}{|\{d' \in D \mid n_{t,d'} > 0\}|}$$

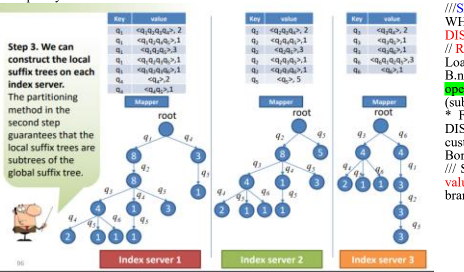
Answering query To answer boolean multi-keyword query such as ("jaguar" AND "new" AND NOT "family") OR "cat" Retrieve inverted lists from all keywords and apply set operations: AND (intersection) ; OR (union) ; NOT (set difference) But there may still be many matched result documents, so we need to define a score for a document d given a query q. score(q,d)= ∑t∈q tf .idf (t,d)

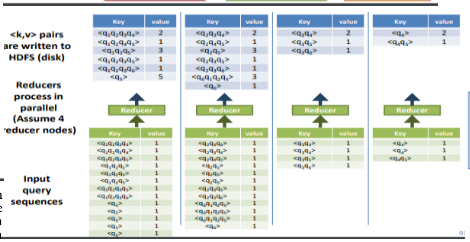


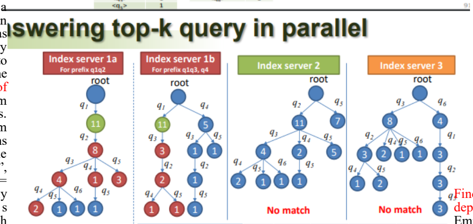
Bound-> 0.37 (0.13+0.24) -> 0.33(0.13+0.20) ->(0.08+0.2) until bound reach maximum, if a score is lower than the one of the result, dun not replace else replace OLAP on Search logs Query suggestion 65 In query suggestion application, the search engine provides suggestions each time the user raises a query. For example, when a user raises a query "Honda", the search engine can apply a forward search function on s1 = <"Honda"> and get the top-k queries as the candidates for query suggestion. Suppose the user raises a second query "Ford", the search engine can drill down using the forward search function to find out the top-k queries following sequence s2 = < "Honda" , "Ford"> as the candidates for query suggestion. Conceptually, a search log is a sequence of queries and click events. Since a search log often contains the information from multiple users over a long period, we can divide a search log into sessions. Sessions extraction Step 1. For each user, extract the queries by the user from the search log as a stream. Step 2. Then segment each user's stream into sessions based on a widely adopted rule : two queries are split into two sessions if the time interval between them exceeds 30 minutes. Data model: E.g., q1="HKU", q2="CUHK", then if a user session s searches for q1 first and q2 next, then s = < q1 , q2 > Session frequency (s) is the number of query sessions that is exactly the same as s. 1. Forward search In a set of sessions, given a query sequence s and a search result size k: The forward search finds k sequences s1 , … , sk such that < s, si > (1 ≤ i ≤ k) is among the top-k most frequent sequences that have s as the prefix. Input s = <"Apple" , "iPhone">, k = 3 Output Top 3 frequent sequences of the form <"Apple" , "iPhone" , <any> > <"Apple" , "iPhone", "iCloud" > <"Apple" , "iPhone", "Apple support" > <"Apple" , "iPhone", "Samsung Galaxy" , "iPad air"> 2. Backward search In a set of sessions, given a query sequence s and a search result size k: The backward search finds k sequences s1 , … , sk such that <si , s> (1 ≤ i ≤ k) is among the top-k most frequent sequences that have s as the suffix. Input s = <"The University of Hong Kong">, k= 3 Output Top 3 frequent sequences of the form < <any> , " The University of Hong Kong"> <"QS Ranking", "The University of Hong Kong" > <"Exchange in Hong Kong" , "HKU", "The University of Hong Kong" > <"Dr. Sun Yat-sen" , "The University of Hong Kong" > 3. Session retrieval In a set of sessions, given a query sequence s and a search result size k: The session retrieval finds the top-k query sessions s1 , … , sk with top session frequency that contains s. Input s = <"Frozen">, k= 2 Output Top 2 frequent sequences that contain "Frozen" <"Disney movie" , "Frozen" > <"Frozen", "Let it go" >



Forward: Input s = <q1q2>, k=2 Result q3:4, q5:3, q4:1 Distributed index structure A search log may contain billions of query sessions. The resulting suffix tree would be a gigantic one that cannot be held into the main memory or even the disk of one machine. A distributed suffix tree construction method under the MapReduce programming model is used. Step 1. Suffixes counting Compute all suffixes and the corresponding frequencies. At the beginning, the whole data set is stored distributively in the cluster. Each compute node possesses a subset of data. Map: For each query session s, the computer emits an intermediate key-value pair (s', 1) for every suffix s' of s, where the value 1 here is the contribution to frequency of suffix s' from s. Reduce: All intermediate key-value pairs having suffix s' as the key are processed on the same computer. The computer simply outputs a final pair (s'; freq(s')) where freq(s') is the number of intermediate pairs carrying key s'. Step 2. Suffixes partitioning Partition the entire set of suffixes into several parts such that each part can be held in the main memory of one index server. Problem: Given a set of suffix sequences (e.g., q1q2q3 and q1q2q4) , how to estimate the size of the subtree created by those suffix sequences? Solution: As we would like to reserve some space in each index server for incremental update, it is okay to obtain a loose estimation. (say, 1+ sum {|s|-1 | for the set of suffix sequences} = 1 + 2 + 2 = 5 nodes.) Map: For each suffix sequence, the computer emits an intermediate key-value pair with key as the prefix query of the suffix, the value retain the suffix pattern and frequency count.



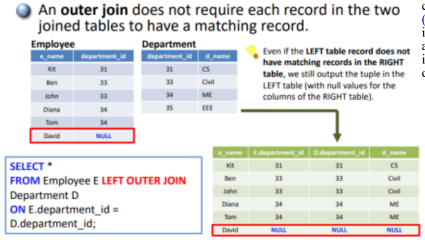




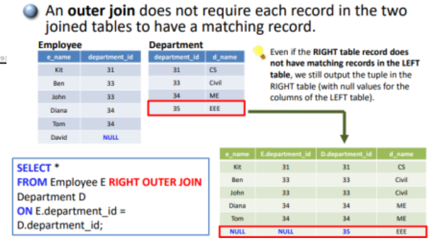
Result from server 1a	q2	q2q3	q2q5	q2q3q4	q2q4
	8	4	3	2	1
Result from server 1b	q3	q3q2	q3q2q4	q3q2q5	
	3	2	1	1	
Combined	q2	q2q3	q3	q2q5	q3q2
	8	4	3	3	2

NOSQL: use company /// db.createCollection('employee')/// show collections/// db.employee.drop()//use company db.dropDatabase()//db.employee.insertOne ({ eID: 1, name: 'Jones', salary: 26000 }) // db.employee.find(); //show dbs // db.employee.insertMany([{ eID: 2, name: 'Smith', salary: 28000 }, { eID: 3, name: 'Parker', salary: 35000 }]) // db.employee.find({},{ eID:1, name:1 }) (get all of eID) // db.employee.find({},{name:0}) (expect name) // db.employee.find({ eID: 1 }) (where eid=1) // db.employee.find({ salary: {$lt: 25000} }) (salary <25000)// db.employee.find ({ $and:[{salary:{$gt:25000}},{salary:{$lte:28000}}]}) (salary > 25000 AND salary <= 28000) // db.employee.find({ eID: {$in : [1,3]} }) (match value in array) // db.employee.find({name:{$regex:'.*mit.*'}}) (SELECT * FROM employee WHERE name REGEXP BINARY '.*s.*';) // db.employee.find({ $or:[{name: {$regex:'^S.*'}}, {salary: {$gte: 30000}}] },{ name:1, salary:1 }) (SELECT name, salary FROM employee WHERE name LIKE BINARY 'S%' OR salary >= 30000) // db.employee.find().sort({ name:-1, salary:1 }) (desc -1, asc 1) //db.employee.updateOne({ eID: 1 },{ $set:{salary:40000} }) // db.employee.updateOne({ eID: 1 },{ $mul:{salary:1.05} })// db.employee.updateOne({ eID: 1 },{ $currentDate:{resign:true} }) // db.employee.updateOne({eID:1},{$unset:{resign:true}}) (remove field) // db.employee.deleteOne({ eID: 5 })// db.employee.find ({ 'department.name':'IT' },{ eID:1, name:1 }) (SELECT E.eID, E.name FROM employee E, works_in W, department D WHERE E.eID = W.eID AND W.dID = D.dID AND D.name = 'IT') // db.employee.find({ 'department.name': { $all: ['IT','Marketing'] } },{ eID:1, name:1 }) // db.transaction.aggregate([{ $group : { _id : null, 'Total num of transactions': {$count: {}} } }]) // db.transaction.aggregate([{ $group : { _id : null, 'Total amount': {$sum: '$amount'} } }]) (replace null by '$customerID') CH2 A foreign key is a referential constraint between two tables. A foreign key is a field in a relational table that matches a candidate key of another table. Author (authorID, name, date of birth) Book (bookID, title, publisher) Writes (authorID, bookID) Foreign key: none Foreign key: none Foreign keys: {authorID} referencing Author {bookID} referencing Book Multivalue attributes (become a table, with foreign key, dun add the attributes to the original table) SQL CREATE TABLE Branch (branch_id VARCHAR(15), name VARCHAR(30) NOT NULL, asset INT UNSIGNED NOT NULL, PRIMARY KEY(branch_id)); // DROP TABLE Branch; //Alter table Add columns to an existing table. ALTER TABLE Branch ADD branch_phone INT (12); Remove a column from a table. ALTER TABLE Branch DROP branch_phone; Add constraints (e.g., PRIMARY KEY) to a table. ALTER TABLE Branch ADD PRIMARY KEY (branch_id); Foreign key constraints CREATE TABLE Owner (customer_id VARCHAR(15), account_id VARCHAR(15), PRIMARY KEY(customer_id, account_id), FOREIGN KEY(customer_id) REFERENCES Customer(customer_id), FOREIGN KEY(account_id) REFERENCES Account(account_id)); /// ALTER TABLE Owner ADD FOREIGN KEY (customer_id) REFERENCES Customer(customer_id); INSERT INSERT INTO Branch VALUES ('B2' , 'Causeway Bay', 9000000), ('B3' , 'Aberdeen', 400000); /// LOAD DATA LOCAL INFILE 'text.txt' INTO TABLE Branch FIELDS TERMINATED BY ';' LINES TERMINATED BY '\n'; // DELETE FROM Branch WHERE name = 'Central'; //UPDATE Branch SET asset = 0 WHERE branch_id = 'B1'; Case UPDATE Account SET balance = CASE WHEN balance <=500 THEN balance *1.05 ELSE balance * 1.06 END SELECT loan_id, amount/7.8 FROM Loan; ///SELECT Borrower.loan_id, Customer.name FROM Customer, Borrower WHERE Customer.customer_id = Borrower.customer_id // SELECT DISTINCT loan_id FROM Loan WHERE branch_id = 'B1' AND amount > 1200; // Rename SELECT DISTINCT Branch.name AS 'Branch name' FROM Branch, Loan WHERE Branch.branch_id = Loan.branch_id ; ///SELECT DISTINCT B.name FROM Branch B, Loan L WHERE B.branch_id = L.branch_id ;// String operations SELECT name FROM Customer WHERE address LIKE '%320%'; (substring) /// '_ _ _%' matches any string of at least 3 characters. //// SELECT * FROM Loan ORDER BY branch_id ASC, amount DESC; //// SELECT DISTINCT customer_id FROM Borrower WHERE customer_id IN (SELECT customer_id FROM Owner); ///// SELECT DISTINCT customer_id FROM Borrower WHERE customer_id NOT IN (SELECT customer_id FROM Owner); /// SELECT AVG(balance) FROM Account WHERE branch_id = 'B2'; (single value) /////SELECT branch_id, AVG(balance) FROM Account GROUP BY branch_id HAVING AVG(balance) >= 650; //// OUTER JOIN

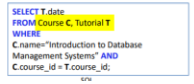
- An **outer join** does not require each record in the two joined tables to have a matching record.

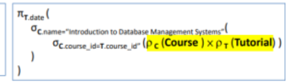


- An **outer join** does not require each record in the two joined tables to have a matching record.

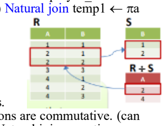


Find the employee_ids and names of employees who work in department with department_id=2. SELECT E.employee_id, E.name FROM Works_in W , Employees E WHERE W.department_id = 2 AND E.employee_id = W.employee_id; Find the dept. names where employee with employee_id = 2 works. SELECT D.name FROM Works_in W, Departments D WHERE W.employee_id = 2 AND D.department_id = W.department_id; : Find the dept. names with at least 2 employee. SELECT COUNT(*), D.name FROM Works_in W, Departments D WHERE D.department_id = W.department_id GROUP BY D.department_id HAVING COUNT(*) >= 2; : Find the employee_id of all employees whose name includes the substring "one". SELECT employee_id FROM Employees WHERE name LIKE '%one%'; Find the employee_id and name of the employees who worked in the departments with budget more than 100,000 SELECT DISTINCT E.employee_id, E.name FROM Works_in W, Employees E WHERE E.employee_id = W.employee_id AND W.department_id IN (SELECT department_id FROM Departments WHERE budget > 100000); : Find the name and budget of the department with the greatest budget. SELECT D2.name, D2.budget FROM Departments D2 WHERE D2.budget = (SELECT MAX(D.budget) FROM Departments D); Find the names of employees who work in at least 2 departments SELECT E.name FROM Employees E WHERE 2 <= (SELECT COUNT(W.department_id) FROM Works_in W WHERE W.employee_id = E.employee_id); In each department, find the highest salary of the employee in that department SELECT MAX(E.salary) , W.department_id FROM Employees E, Works_in W WHERE E.employee_id = W.employee_id GROUP BY W.department_id; UNION INTERSECT EXCEPT Find department names that have greater budget than some department where employee 4 works. SELECT D.name FROM Departments D WHERE D.budget > SOME(SELECT D2.budget FROM Departments D2 WHERE D2.department_id IN (SELECT W.department_id FROM Works_in W WHERE W.employee_id = 4)); (ALL, if empty, all is true, some is false) Find the names of employees who work in department with department_id=1. SELECT E.name FROM Employees E WHERE EXISTS (SELECT * FROM Works_in W WHERE W.department_id = 1 AND E.employee_id = W.employee_id); (IS NULL) count(*) include null value ////CREATE VIEW Employee_hide_salary AS (SELECT employee_id, name FROM Employees); GRANT SELECT ON Departments TO Johnson, Brown CREATE ROLE manager; GRANT manager TO Brown;/// CREATE ASSERTION EmpsNoLessThanDepts CHECK ((SELECT COUNT(*) FROM Departments) <= (SELECT COUNT(*) FROM Employees)); List the iID, iname, usageDec of the ingredient(s) in the recipe of the dish with dID as 2. SELECT DISTINCT I.iID, I.iname, R.usageDec FROM Recipe R INNER JOIN Ingredient I ON R.iID = I.iID AND R.dID = 2 ORDER BY I.iID DESC; 3.List the iID, iname of the ingredient(s) in the recipe of the Chinese cuisine dish(es) with "crab" appear in the introduction. SELECT DISTINCT I.iID, I.iname FROM Ingredient I INNER JOIN Recipe R ON I.iID = R.iID INNER JOIN Dish D ON R.dID = D.dID WHERE D.introduction LIKE '%crab%' AND D.cuisine = 'Chinese' ORDER BY I.iID DESC; 7.List the iID, iname of the common ingredient(s) among the dishes ("Garlic Crab Legs", "Crab Salad", and "Fried Shrimp") SELECT DISTINCT I.iID, I.iname FROM Ingredient I INNER JOIN Recipe R ON I.iID = R.iID INNER JOIN Dish D ON R.dID = D.dID WHERE D.dname IN ('Garlic Crab Legs', 'Crab Salad', 'Fried Shrimp') GROUP BY I.iID, I.iname HAVING COUNT(DISTINCT D.dID) = 3 ORDER BY I.iID DESC; CH5 σ name="May"(Author) π bookID, title(Book) πname (Audio_CD) ∪ πname (DVD) Set difference πstudent_id(Student) – π student_id(Submit), have same number of attributes. Cartesian product





Set intersection (same number of attribute) πemployee_id (σW1.department_id=1 ∧ W2.department_id=3 (σW1. employee_id =W2. employee_id (ρW1 (Works_in) × ρW2 (Works_in)))) Natural join temp1 ← πa (R) ρResult(branch_id, sum_of_balance) (branch_id g sum(balance) (Account))



Equivalence rules
1.Only the final operations in a sequence of projection operations are needed.
2. Conjunctive selection operations can be deconstructed into a sequence of individual selections. σp1 ∧ p2(E) = σp1 (σp2 (E)) 3. Selection operations are commutative. (can change place) σp1 (σp2 (E)) = σp2 (σp1 (E))4. Natural join operations are associative. (E1⋈ E2) ⋈ E3 = E1⋈ (E2 ⋈ E) Rule 5. The selection operation distributes over the natural join operation under the following two conditions. Rule 5a. It distributes when all the attributes in selection condition involve only the attributes of one of the expressions (say, E1) being joined. σp (E1⋈ E2) = (σp (E1) ⋈ E2) Rule 5b. The selection distributes when selection condition p1 involves only the attributes of E1 and p2 involves only the attributes of E2 . σp1 ∧ p2(E1⋈ E2) = (σp1 (E1) ⋈ σp2 (E2)) Rule 6. The projection operation can distribute over the natural join operation. πL1 ∪ L2 (E1⋈ E2) = πL1 ∪ L2 ((πL1 ∪ L3 (E1)) ⋈ (πL2 ∪ L3 (E2))) Rule 7. The set operations union and intersections are commutative Rule 8. The set operations union and intersections are associative Rule 9. The selection operation distributes over the union, intersection and set difference operations Rule 10. The projection operation distributes over the union operation