**Multi-key Processing**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Goal**  A particular table in a database may need to be queried based on different selection criteria. We need to understand how best to represent our data to allow a variety of queries to perform well. | **Example 1**: STUDENT Table   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | ST# | Name | Classification | Major | GPA | GPHours | GPTot | | Primary Key | Fairly Unique | FR,SO,JR,SR,GR | CS,MAT,  ENG,HIS |  | Grade Point Hours | Grade Point Total |   We expect queries based on   * ST# * Name * Classification – (e.g., get all SR) * Major * Classification and Major – (e.g., get all SR CS majors) * Classification and GPA |
| **File structure considerations**  We need to represent the primary key and data. This may involve   * hashing * indexing – usually with a B+Tree * clustering of the data in a manner based on some index   We also have non-primary key accesses. Those are impacted by the primary key, data representation, and secondary key implementations   * linked list * inverted list * indexing – B+Tree | Primary Key and Secondary Keys can be single attributes or composites of two or more attributes. |
| **Indexing**  Most indexes use B+Tree implementations. With B+Trees, we have an index area which may have many index levels. The entries in each node (except the bottom index level) has the highest keys for the referenced nodes.  The data level can be **clustered** or unclustered. With clustering, we *attempt* to place the data near where it belongs in order by the index. With unclustered, the data is considered to be in a *random* order.  Clustering doesn't always have to be based on primary keys; instead, it may make more sense to base it on a secondary key. | **Example 2:** Student data clustered by Student Number. |
| **Clustering by a Non-Primary Key**  Suppose we have an **Employee** table which contains Emp#, Name, Salary, OrgUnit, and other attributes. We have the following selection criteria in our queries:   * Emp# - access a particular emp * OrgUnit - access employees in a particular OrgUnit.   On the average, there are 10 employees per OrgUnit. We have 9000 employees.  It is usually better to cluster based on a frequent query that returns many rows.  Note: that a query that does sorting (ORDER BY in SQL) may also benefit from clustering.  We can have **ONLY one clustering order**. | **Example 3**: clustering by Emp# vs Clustering by OrgUnit (then Emp#)  Assume:   * The index has three levels. * We place 15 employees in each data block. * We have **two indexes**: * unique index on Emp#. Each index node contains 50 entries.   + Lowest level has 180 index nodes(9000 emp/ 50ent per blk)   + Next level has 4 index nodes(180 ent / 50ent per blk)   + One root node containing 4 entries * unique composite index on unit# and Emp#. We have 25 entries per index node   + Lowest level has 360 index nodes (9000 emp / 25 ent per blk)   + 15 index nodes on the next level (360 / 25)   + One root node containing 15 entries   **If we cluster on Emp#:**   * To access an employee by emp#:   + We use the Emp# index.   + It takes 3 index block reads since there are 3 index levels   + 1 data block read   + **Total of** 4  **reads to get one employee by emp#** * To access all of the employees in a unit:   + We use the composite index.   + Since the second index is ordered by OrgUnit (then Emp#), the index entries are in order by OrgUnit. Lowest level should have them in either 1 or 2 index blocks. We will assume 1 at the lowest level. Total of 3 index block reads.   + For the data which isn't ordered by unit, we will probably rarely have employees in the same data block. We will have to do a data read for each of the 10 employees. (on the average in a unit)   + **Total of 13 reads to get all of the employees in a unit (on the average).**   **If we cluster on the combination of OrgUnit and Emp#:**   * To access an employee by emp#:   + We use the Emp# index.   + It takes 3 index block reads   + 1 data block read   + **Total of** 4 **reads to get one employee by emp#** * To access all of the employees in a unit:   + We use the composite index.   + Since the second index is ordered by OrgUnit (then Emp#), on the average we will find Lowest level should have them in either 1 or 2 index blocks, We will assume 1 at the lowest level, Total of 3 index block reads.   + For the data which is ordered by OrgUnit (then Emp#), most of the entire unit can be found in the same data block; however, the employees from one unit will be in either one or two blocks. (About ¼ of the time they will be in just one block) We will make that approx 2 reads.   + **Average of** 3 index reads + 2 data reads = 5 total **reads to get all of the employees in a unit.** |
| **Unique vs Duplicate keys in B+Tree**  If our B+Tree implementation only supports unique keys, we can concatenate the primary key onto the end of the secondary key making it unique. This does have the advantage of being able to satisfy some queries without requiring access to data blocks (e.g., get each of the Emp# for a given OrgUnit). | **Example 4**: B+Tree with duplicate keys vs unique keys  Many internal implementations of B+Trees use unique keys although the user may think they support duplicates. In example 3 for an index on OrgUnit, the implementation could be either  1. Specify that OrgUnit has duplicates. The internal implementation could make it unique by simply including the data pointer as part of an internal B+Tree key.   * OrgUnit * RowId   This also helps with clustering since the entries are ordered by the data block (due to the RowId)  2. Specify that the key is unique and is a combination of OrgUnit and Emp#. This makes entries unique, but the ordering might not be as good as the combination of OrgUnit and RowId in reducing input operations.  It is worth finding out how a particular DBMS implements duplicates to determine whether we should concatenate the primary key or whether it concatenates the RowId. |
| **Secondary Key Implementations**  **B+Tree** provide a B+Tree on the secondary key value  **Linked List** link together rows having the same key value  **Inverted List** separate list of data pointers for a common secondary key value  **Hashing** hash table of key and data pointer | Hashing is typically used for unique or fairly unique values. It doesn't support partial key searching. |
| **Linked List**  Provide a linked list for each value of the secondary key. For classification, include a linked list for each of these: FR, SO, JR, SR, GR. The lists required a control record which contains each of the linked list heads.  To add a new value for Classification (e.g., BE for Basic Entry), we have to add a new BE\_HEAD to the linked list.  If we want to create linked lists for Major, we have to add a new column to the table for the Major LL, and we have to add the Major Heads to the control.  Would it make sense to use linked lists for Name?  ? | **Example 5**: linked lists on Classification. Assume the table was stored using hashing on the primary key (ST#).  Control would contain:  FR\_HEAD 8  SO\_HEAD 9  JR\_HEAD 3  SR\_HEAD 4  GR\_HEAD 32   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **Rec** | **ST#** | **Name** | **Classif** | **Classif**  **LL** | **Major** | **GPA** | **…** | | 1 | 7 | Sue | JR | 10 | MAT | 3.9 |  | | 2 | 3 | Bob | SR | 7 | CS | 4.0 |  | | 3 | 10 | Joe | JR | 1 | ENG | 3.6 |  | | 4 | 25 | Cass | SR | 6 | CS | 4.0 |  | | 5 | 5 | Kris | SR | 18 | CS | 3.9 |  | | 6 | 12 | Paco | SR | 2 | CS | 4.0 |  | | 7 | 20 | Will | SR | 12 | CS | 3.8 |  | | 8 | 30 | Raj | FR | 41 | MAT | 2.9 |  | | 9 | 15 | G | SO | 11 | MKT | 2.5 |  | | 10 | 35 | Shane | JR | 15 | MAT | 2.7 |  | | 11 | 50 | Hugh | SO | 37 | MAT | 3.2 |  | | … |  |  |  |  |  |  |  | |
| **Inverted List**  For each value of the secondary key, maintain a list of pointers (possibly record numbers or data pointers). This is kept in a separate file or portion of a file.  If we want to create an inverted list for major, we add another separate file on major.  If the cardinality of Classification values is low, we can simply include them in a block (a starting block for each value), with long lists of pointers, we can place them in additional blocks. | **Example 6: inverted list**  Inverted List on Classification   |  |  | | --- | --- | | Value | Record Numbers | | FR | 8 41 42 45 50 … | | SO | 9 11 22 24 37 38 39 … | | JR | 1 3 10 15 16 17 21 … | | SR | 2 4 5 6 7 12 20 25 27… | | GR | 32 … | |  |  |   Inverted List on Major   |  |  | | --- | --- | | Value | Record Numbers | | CS | 2 4 5 6 7 21 22 27 … | | ENG | 3 12 18 20 … | | MAT | 1 8 10 11 16 25 … | | MKT | 9 13 15 17 … | |
| **Queries Involving Multiple Secondary Keys**  How can we satisfy queries involving multiple secondary keys?  With linked lists, we really can use the list for only one value. We can then examine data on the row for the other value.  With inverted lists, we can get the intersection of two inverted lists fairly cheaply and then read those rows.  Solution #3: Intersect two inverted lists (one on Major and the other on Classification)   * Assume 500 entries per block in the inverted list * One read to get the list of Majors in the inverted list * Two reads to get the list of Seniors. * The intersection is done in memory. * We expect 2% of 1000 seniors to be ENG majors; therefore, 20 students. * Since it is very unlikely for two ENG students to be in the same data block, it would take us 20 reads in the data area * Total = 1 + 2 + 20 = 23 reads | **Example 7**: get the ENG Majors who are SRs. How many reads?  Assumptions:   * 10 students per data block * 1000 SRs * 200 ENG Majors * 10,000 students * linked list is ordered by data block * inverted list is ordered by data block   Solution #1: Linked List on Classification   * percentage of students who are ENG majors = 200 / 10,000 = 2% * Out of 10 students per data block, it is very unlikely that there will be more than one ENG student in the data block. * percentage of students who are seniors = 1000/ 10,000 = 10% * On the average, there will be one SR per data block. * It will take us 1000 reads using a linked list on SR. For each of those, we check the MAJOR column for ENG.   Solution #2: Linked list on Major   * percentage of students who are ENG majors = 200 / 10,000 = 2% * Out of 10 students per data block, it is very unlikely that there will be more than one ENG student in the data block. * It will take us 200 reads using a linked list on ENG. For each of those, we check the CLASSIFICATION column for SR. |
| **Comparison of Linked Lists vs Inverted Lists**  **Linked Lists**  **Adv:**   * Easy to process   **Disadv**:   * Row order in the linked list may require jumping to different data blocks for every row * Inefficient for unique keys * Difficult to add/delete a secondary key since that would require adding an new linked list attribute to the row. * Queries involving multiple separate secondary keys can only follow one list instead of obtaining an intersection of data row pointers * Cannot do an efficient partial key match as can be done with B+Trees | **Inverted Lists**  **Adv:**   * Unique keys are easily processed * Queries involving multiple separate secondary keys can be processed more efficiently by merging the results * Effective adding/deleting of secondary keys   **Disadv**:   * Maintain variable-length index records * More complex data structure than linked list * Cannot do an efficient partial key match as can be done with B+Trees |
| **B+Tree with a duplicate composite key on MAJOR then CLASSIFICATION**  It is easy in a B+Tree implementation to support full key and partial key searching. On that composite key, we could easily search on the combination or search only on the left most portion (MAJOR). | **Example 8**: get the ENG Majors who are SRs using a composite on MAJOR then CLASSIFICATION. How many reads?  **Assumptions**:   * 10 students per data block * 1000 SRs * 200 ENG Majors * 10,000 students * Overhead per index block is 16 bytes. * RowIDs are 8 bytes * Major is 3 bytes, Classif is 2 bytes. * Data is clustered by StudentId.   **Solving**   * Number of entries per index block in the composite key index = ?.   + 4096 - 16 for overhead = 4080   + 3 from major, 2 from classif, row id = 8   + Size of entry = (3+2+8) = 13 bytes per index entry   + 4080 usable bytes per index block / 13 bytes per entry  = 313.8 -> 313 entries per block * Number of bottom level index blocks = 10000 entries / 313 entries per block  = 31.9 -> 32 blocks * Next level   + **Each block below would have an entry**   + 32 entries /313? entries per block =1 index block. We have 2levels. * How many ENG SR do we expect?   + What percentage of students are SRs? 10%   + We expect 20 students to be ENG SR (10% of 200 ENG majors) * All of those ENG SR will most likely be in the same index block. Reads in index = 1(root) +1 (lowest level) reads, (since 20 ENG SRs would usuall fit in 1 index block) * Since our clustering index is studentId not our composite index, it is very unlikely for two ENG students to be in the same data block, it would take us 20 reads in the data area * Total = 2 + 20 = 22 reads |
| **For the bottom level in an index, what does the data pointer contain?**  Since a row may contain several variable-sized attributes, a row might grow or shrink. If index entries were to be the actual byte address of rows, we would have to change the index entry each time the row changes in size.  To avoid having to alter an index block reference, most implementations use a data pointer that contains:   * **data relative block number** - simply an integer representing which data block * **line array subscript** - a subscript within an array in the data block. The corresponding entry contains a reference to the location of the row within the block.   This allows many indexes to point to a data pointer without having to be modified if the row moves around within a data block.  Some implementations also don't change the index entries if the row no longer fits in the data block. Instead, the line array reference would point to a indirect row reference within the current data block. The indirect row reference would contain a data pointer to the data block containing the row. | **Example 9:** What is a data pointer? |
|  |  |
|  |  |

**©2020 Larry W. Clark,** UTSA CS students may make copies for their personal use