CS 4420 DATABASE PROJECT (Option 1)

Spring Semester 2004

The overall objective of this project is to give you a better understanding about some of the internals of a database management system. To do this you will build a simple single-user database system that will execute a restricted form of SQL. By simple here, we mean that many features of a real relational DBMS will be simplified to make this project manageable as a course project.
The project will be implemented in two phases. The first phase will involve writing the *Storage Manager* component of a database system. The *Storage Manager* is responsible for providing different file structures such as sequential files and indexed files to store data, providing access to the files (insert and fetch, we will not worry about delete) and providing/managing the main memory buffer pool. For our case, the *Storage Manager* will use the basic services provided by Unix (e.g., lseek, read, write) for carrying out low level data transfer between disk and the buffer area. In phase I, we will also include the construction of a system catalog which will include the names of relations (i.e., files), columns in each relation, data type, etc. The second phase will involve writing the *Query Engine* for our database system. The *Query Engine* consists of several components: a scanner/parser/validater, a query optimizer, an access plan generator and a runtime database processor. The scanner/parser/validater identifies the tokens in the text of the query, checks the query syntax and validates relation and attribute names and produces a query tree. The query tree is passed to the query optimizer which uses some basic rules and optimization strategies to rewrite the query tree so that it can be executed in a more efficient manner. The optimized query tree is passed to the access plan generator that provides the algorithms for executing the specific relational algebra operations in the order specified in the tree. The access plan generator uses statistics contained in the system catalog for determining the specific algorithms and access strategies to use (e.g., file scan or index scan). Once the access plan has been determined, the runtime database processor will execute this query-specific access plan to produce the result for the query.

The project has two milestones:

Milestone 1: A report on Phase I design and implementation will be produced. The **due date** is the Thurday of the eighth week of the course, which is **February 26, 2004**. You may hand in your report in class on that day or email it to instructor and TA.

Milestone 2: the software package of the entire product of your project and a final report will be produced. Each project team will schedule a demo date and time to demonstrate their product and walkthrough the code. The **due date** of your final project is the midnight of the Thursday of the last week of the semester, which is **April 22, 2004**. You should send

in a compressed tar file including the source code with good documentation, the printout of your data used in the project, some sample screenshots, and the final report.

A detailed description of the deliverables for each of the two milestones will be made available online at the course web site.

PHASE I:

The *Storage Manager* will be responsible for handling

- the creation of relations and indexes, i.e., entering their definition into the system catalog and for each relation creating the header block for the file storing the relation;
- inserting records into each newly created file, records will be entered sequentially on disk;
- inserting key values and addresses into the index;
- fetching a page from the disk (hint: a simplification is to use one block per page. The description of blocks are provided in Phase I description).

The file structure for the relations will be a sorted sequential file. The records will be given as input in the necessary order so no sorting will be required. In addition, a file will be ordered on the first attribute (i.e., column) value in ascending order.

The system catalog will reside on disk as will the files storing the relations. The catalog will contain the following information:

- the name of a relation (character string of length 4 maximum)
- an internal relation ID number
- number of tuples in a relation
- the indexed column(s)
- the index file name(s) (character string of length 4 maximum)
- for each column of a relation,
  - column name (character string of length 4 maximum)
  - an internal column ID number
  - data type for column (suggestion: all indexed columns are restricted to integer)
  - number of distinct values in a column

Note: A relation will typically have many columns and the columns will be listed in order within a tuple. In this project we suggest that you consider at most two indexes on individual columns for

each relation. You will find additional catalog information by carefully reading the requirements of PHASE II.

The file structure for a relation will be an ordered (sorted) sequential file. We will access a particular block (i.e., a page) by its offset within the file. The file will be initially sorted by the first column, so a *clustered* index may be created for this column. We may also create one secondary index on some other column. Both B tree and B+ tree code will be provided. Examining the code will give you some insight into how to create/insert into the data file. After you create the data file for the relation you can extract the necessary information that will be inserted into the index file.

The basic operations on the data file will be

- create the file - insert an entry into the system catalog and create a header block for each disk file
- insert a record - place the new record in last block of file
- fetch a single block - read a block based on its offset within the file

The basic operations on the index file will be

- create the index - create an entry in the system catalog and create a header block for each index file
- insert a (key, address) pair - the address is just a relative block number of where the record resides with the given key value
- search the index - returns the relative block number (location) of the record with the given key value. In the case of a secondary index (where duplicate keys are allowed), we need to access the index block containing pointers to data blocks that contain records with the same key value (at most one block of pointers will be needed). A pointer gives the relative block number of a record matching the given key value.

Another component associated with the *Storage Manager* is the *Buffer Manager*. The *Buffer Manager* is responsible for maintaining a main memory pool of blocks (either data, index or catalog). You may view the buffer pool as a cache. However, in this project, the *Buffer Manager* will only be concerned with data blocks. When a data block is needed the *Buffer Manager* is called. It will check if the main memory buffer pool contains the desired block. If so the main memory address is returned. If the desired block is not in the buffer pool, a physical I/O (i.e., a READ) must be issued to retrieve the block from disk. The retrieved block will be stored in a free location in the buffer pool and its main memory address will be returned. If there is no free block, the *Buffer Manager* will use a *Least Recently Used* buffer replacement algorithm to find a block in the buffer that will be replaced by the new incoming block. Usually, the *Buffer Manager* maintains an internal hash

table or index table for fast lookup in the buffer pool. This table associates a buffer location with the disk blocks (identified by file ID and relative block number within the file) currently in the buffer. The *Buffer Manager* will also maintain some statistics about the number of logical block accesses (the number of requests for the data block) and the number of physical block accesses for data (the number of disk fetches performed). Every time the *Buffer Manager* gets a request for a block, it increments the logical block access counter and if the requested block is not in the buffer, then it will also increment the physical block access counter.

The *Storage Manager* will accommodate two basic access paths: *Table Scan* and *Index Scan*. For a *Table Scan*, the *Storage Manager* will be called to open the associated file. Then the *Storage Manager* will be called repeatedly to return each block of the file in physical order. For an *Index Scan*, the *Storage Manager* will be called to open the associated index file. Then the *Storage Manager* will be called to find the address associated with a given key value. The block address will be returned. The block may be an address of a data block or an address of a block containing pointers to data blocks that contain records with the same key value. Finally the *Storage Manager* is called to open the associated data file (if not already open) and to read the necessary data block. The value returned by the *Storage Manager* is the main memory address of a block in the buffer pool. As previously mentioned, the requested block may be found in the buffer, thus eliminating a disk access.

The block size we will use is 512 bytes. The record size for the relations will be determined by the number of columns (4 bytes per column). There will be no more than 10 columns in any table. Records should not be split between blocks. We will have a buffer pool of 25 blocks.

For *PHASE I*, your system should accept as input the following database language commands:

CREATE TABLE $RelName(AttrName_1, typespec_1[, AttrName_2, typespec_2, \ldots, AttrName_{10}, typespec_{10}])$;
CREATE INDEX $IndexName$ ON $RelName(AttrName_i)$ [NO DUPLICATES];
LOAD TABLE $RelName\ datafile$;
LOAD INDEX $RelName\ IndexName$;
LOAD RSTATS $RelName\ statisticaldata$;
LOAD ISTATS $RelName\ IndexName\ statisticaldata$;
PRINT TABLE $RelName$;
PRINT INDEX $RelName\ IndexName$;
PRINT CATALOG;
PRINT BUFFERSTATS;
RESET BUFFERSTATS;

Note: The commands are terminated by a semicolon and command keywords are case-sensitive (i.e., capital letters). Square brackets in the syntax specifications indicate command components that are optional. For the CREATE TABLE command, relation and attribute names are limited to 4 characters each and must begin with a letter. Every relation name must be unique and within a given relation, every attribute name must be unique. A relation will have a maximum of 10

attributes. To simplify things, all attributes will be restricted to 4 bytes in length and two data types Integer or String. All the rows will be inserted into a table at one time via the LOAD TABLE utility. For the CREATE INDEX command, the associated catalog information for the index is created. At most two indexes may be created for a single relation on different attributes. The default is that duplicate values may occur for the indexed attribute. If duplicates cannot occur, then NO DUPLICATES should be specified in the CREATE INDEX statement. The LOAD INDEX utility will create the index file on disk. It will be responsible for reading the relation and storing the key value/address pairs in the index file. The LOAD RSTATS utility is responsible for putting the statistical information about the tables and columns into the catalog. The LOAD ISTATS utility is responsible for putting the statistical information about the indexes into the catalog. The three PRINT commands are also utility functions but are only needed to help verify that your PHASE I is working. The PRINT TABLE command will print the contents of a relation, one row per line. The PRINT INDEX command will print the key values in the leaf nodes of the index along with their data block addresses. The PRINT CATALOG command will print the contents of the system catalog. The PRINT BUFFERSTATS command will print the contents of the two block access counters. The RESET BUFFERSTATS command will reset the contents of the two block access counters to 0.

B+ Tree Information:

The B+ Tree index file will reside on disk. Each B+ Tree node uses one disk block. The first block of the B+ Tree index file stores a header block. The header stores the block number (disk address) of the root node in the first 4 bytes and the block number of the next available block. The remainder of the first block in the B+ tree index file is unused.

Blocks are allocated for the index starting with the second block in the file. When a new block is needed, the file will be extended by one block. All key values and pointers are 4 byte integers. The remaining 4 bytes in a node is used to hold a flag, which indicates whether the node is a leaf node or a non-leaf node, and a count of the number of key values stored in the node. For leaf nodes, the pointer to the next leaf node in the sequential chain is stored in Ptrs[0]. A value of -1 indicates the end of the chain. The pointer for Keys[k] is in Ptrs[k+1]. If duplicates are allowed, then the a pointer in a leaf node will point to a bucket containing pointers to blocks in the data file.

IMPORTANT: As a milestone of the project, we require that each project team produce a Phase I report documenting (1) the set of relations and indexes created; (2) the set of functions provided by your Storage Manager; and (3) the list of simplifications made.

Due date: Thurday of the 8th Week, which is February 26 2004.

PHASE II:

The *Query Engine* will consist of 4 basic components:

- Scan/Parse/Validate: The input is broken down into tokens, query syntax is checked and

validated and a query tree is produced. To check the syntactic correctness, we need to verify that relation names and attribute names are correct as found in the catalog and that query syntax follows the grammar rules. The query tree is the internal representation for the query and is simply a binary tree where the leaves represent relations and internal nodes represent operations.

- Query Optimizer: The query tree is used along with the optimizer rules that are included in the query optimizer to rearrange the tree into an "optimized" query tree. The basic optimization strategy we will use is to push project and select operators as far down in the tree as possible. The only other optimization we will use is to order the inner and outer relations of a join which will be decided in the next step.

- Query Code Generator: The optimized query tree is used along with statistics contained in the catalog to determine the "optimal" plan for executing the query tree. For example using an index for a selection operation or reordering the operand relations in a join operation. An *Access Plan Table* will be constructed to hold this information. This table will also be written to disk (in ASCII format) so that its contents can be examined/printed later. That way we can see if the optimizer is working correctly. The contents of the *Access Plan Table* looks like the following:

  - operator: either select, project or join
  - conditions/attributes: list of (attribute, comparison operator, value) triples if operator is a select or list of attribute(s) if operator is a project, or the pair of join attributes for a join operator
  - table1: outer (left) relation operand for operator
  - table2: inner (right) relation operand for operator, only applies if operator is a join and represents the inner relation for the join
  - table1 access method: either index scan or table scan for select, only a table scan for a project or join
  - table2 access method: applies only for join operator and is either an index scan or table scan
  - name of index: name of index used (if any)
  - result table: name of temporary table on disk which stores the result of the specified operation

  Inherent to this step is the ability to estimate the size of the relations that result from various operations. We will use the notion of filter factors (also referred to as selectivity factor) to accomplish this. The filter factor is defined for different predicate types and can be combined to determine the filter factor of predicates combined by *OR* and *AND*. Although our queries

will only have predicates combined by *AND*. The filter factor of a predicate P, denoted by FF(P), is defined as the fraction of rows from a table resulting from the predicate restrictions. The filter factor of a predicate is estimated by making a number of statistical assumptions. Typical assumptions used in most of RDBMSs are the following two: individual column values are uniformly distributed and values from any two columns are independently distributed.

Necessary statistical information includes the following:

TABLES

Statistic Name	Description
CARD	# rows in table
NPAGES	# pages holding rows

COLUMNS

Statistic Name	Description
COLCARD	# distinct values
HIGHKEY	highest value
LOWKEY	lowest value

INDEXES

Statistic Name	Description
NLEAF	# leaf pages
NBUCKET	# buckets
KEYCARD	# distinct values in key

The basic filter factor formulas are as follows:

Predicate Type	Filter Factor (FF) Formula
C1 = Value	$1/COLCARD$
C1 > Value	$(HIGHKEY - Value)/(HIGHKEY - LOWKEY + 1)$
C1 < Value	$(Value - LOWKEY)/(HIGHKEY - LOWKEY + 1)$
Pred1 AND Pred2	$FF(Pred1) * FF(Pred2)$

Formulas for estimating the join size of $R(X,Y) \bowtie S(Y,Z)$ are shown below. If the join attribute, Y, is a key for relation S, then each tuple of R will join with at most one tuple of S, so we can estimate the join size as $CARD(R)$ tuples. For other situations we will use the following:

$$\frac{CARD(R) \cdot CARD(S)}{Max(COLCARD(R,Y), COLCARD(S,Y))}$$

where $COLCARD(R,Y)$ is the number of distinct Y values in relation R and $COLCARD(S,Y)$ is the number of distinct Y values in relation S.

To determine the most efficient access plan, the statistics and filter factors will be used to

estimate the I/O cost of the query. In general, to determine the number of I/Os needed, the filter factor is multiplied by the number of rows or pages in the table. To estimate the number of I/Os needed for a table scan, it is simply the NPAGES statistic. To estimate the number of I/Os for an index scan, it is dependent on whether the index is clustered (i.e., the data is clustered on the indexed attribute). We will ignore the cost of accessing internal nodes in the index. If the index is clustered then we simply have $FF(Pred) * NLEAF \ + \ FF(Pred) * NPAGES$. If the index is not clustered, then we have $FF(Pred) * NLEAF \ + \ FF(Pred) * CARD$. We can estimate $NLEAF$ as $\frac{2*KEYCARD}{ORDER-1}$, where $ORDER$ is the maximum number of pointers per node and the factor of 2 is included since we assume that the leaf nodes will be only half full. If duplicates are allowed then we also have to include the access cost to the buckets (i.e., pages containing pointers to records with the same key value). So, we need to add $FF(Pred) * NBUCKET$. We can estimate $NBUCKET$ as $\frac{CARD}{2*ORDER}$.

Note: At most one index will be used in processing a *SELECT* operation.

To estimate the size of the result of a unary operation, we can use $FF(Pred) * CARD$ as the number of resulting tuples. We divide this by the blocking factor (i.e., number of tuples per block) to determine the size in blocks.

- Runtime Database Processor: The access plan table is processed in order and the operations/algorithms are executed. Temporary results are stored as a relation on disk, so the catalog must be accessed and a new file created. The result of the query will also be written to disk and then read back and displayed on the screen.

Design of the first component

The query language command looks like the following:

SELECT $A_1[, A_2, \ldots, A_m]$
FROM $RelName_1[, RelName_2, RelName_3]$
[WHERE A'_1 Op_1 AV_1 AND A'_2 Op_2 AV_2 AND ... AND A'_5 Op_1 AV_5];

Note: Each A_i in the SELECT clause is specified as either *RelName.AttrName* or *AttrName*. The SELECT clause must have 1 or more attributes listed. Each $RelName_i$ in the FROM clause must name a relation in the database and a relation may not appear more than once in the FROM clause. Also, at least 1 relation and at most 3 relations can appear in the FROM clause. For each *RelName.AttrName* in the SELECT clause, if *RelName* is specified, then it must be one of the relations in the FROM clause. IF *RelName* is not specified, then there must be exactly one relation in the FROM clause with an attribute named *AttrName*. The same rule applies for the WHERE clause. Each A'_i in the WHERE clause is specified as either *RelName.AttrName* or *AttrName*. Each AV_i in the WHERE clause is specified as *RelName.AttrName* or *AttrName* or it is a constant value (an integer). Each Op_i in the WHERE clause is one of the comparison operators, $=, <, >$ if AV_i is a constant, otherwise Op_i can only be $=$ (i.e., representing an equijoin).

The WHERE clause is optional, but if it is present it must have at least one comparison term (i.e., $A'_i\ Op_i\ AV_i$) and at most 5 comparison terms, separated by AND.

Using our database language, we could define the following tables and indexes:

```
CREATE TABLE r1 (A, B, C);
CREATE TABLE r2 (C, F, G, H);
CREATE TABLE r3 (A, D, E);
CREATE INDEX r1AX ON r1(A) NO DUPLICATES;
CREATE INDEX r2CX ON r2(C) NO DUPLICATES;
CREATE INDEX r2FX ON r2(F);
CREATE INDEX r3AX ON r3(A) NO DUPLICATES;
```

and associated queries

SELECT F, G FROM $r2$;
SELECT C FROM $r1$ WHERE $A > 100$;
SELECT A FROM $r1$ WHERE $B = 7$ AND $C = 5$;
SELECT A, H FROM $r1, r2, r3$ WHERE $B > 7$ AND $E = 3$ AND $r1.C < 3$
AND $r1.C = r2.C$ AND $r1.A = r3.A$;

We will assume that the ordering of relations in the FROM clause infers a particular join ordering for those relations.

Consider the relations: R1(A,E,X), R2(B,F,G,W) and R3(C,H,Y,Z), and the query:

```
SELECT A,B,C
FROM R1,R2,R3
WHERE R1.E = R2.F AND R2.G = R3.H;
```

The FROM clause has the particular order R1,R2,R3 which we assume to mean R1 can join with R2 and R2 can join with R3.

The query tree produced by SCAN/PARSE/VALIDATE is shown in Figure 1.

Design of the second component

The second part of PHASE II, i.e., QUERY OPTIMIZER, will optimize the tree by doing the following in order:

1. Pushing Selections down the tree
2. Choosing the order of JOINs
3. Choosing the outer relation for a JOIN
4. Pushing Projections down the tree

Because of the restriction we placed on the FROM clause, there are only 2 different JOIN orderings to consider, which are $(R1 \bowtie R2) \bowtie R3$ and $(R2 \bowtie R3) \bowtie R1$. After doing the first three steps, the

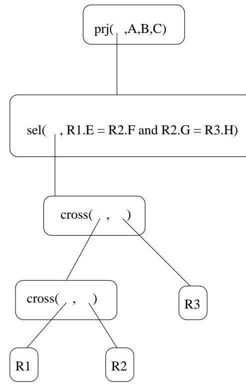

Figure 1. Query Tree

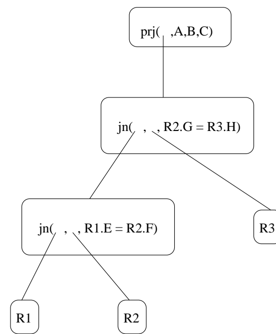

Figure 2. Partially optimized query tree

tree appears as shown in Figure 2 (assuming that the estimated size of $(R1 \bowtie R2)$ is less than the estimated size of $(R2 \bowtie R3)$ and that the size of R1 is less than the size of R2. We will assume the the left child of a JOIN node is the Outer relation for the join operation, which should be the smaller of the two.

We should note that choosing the order of JOINs and the outer relation for a JOIN involve using the statistics from the catalog and the formulas for estimating sizes of results of relational operations. After doing the fourth step, we have the final optimized tree as shown in Figure 3.

When we JOIN two tables, we will assume that there is only one column from each relation which is used in the equi-join operation.

Design of the third component

In part 3, QUERY CODE GENERATOR, we will only consider one join algorithm, i.e., the NESTED-BLOCK algorithm, where we fill the buffer with a number of blocks from the Outer

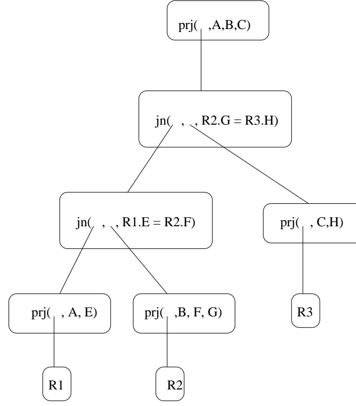

Figure 3. Final optimized query tree

	Op	Con/Attr	Tab1	Tab2	Tab1 Access	Tab2 Access	Index name	Result Tab
(1)	sel	$A = 100$	r1		index		r1AX	t1
(2)	prj	$B\ C$	t1		table			t2
(3)	sel	$F = 5$	r2		index		r2FX	t3
(4)	prj	$C\ G$	t3		table			t4
(5)	jn	$C\ C$	t2	t4	table	table		t5
(6)	prj	$B\ G$	t5		table			t6

Table 1: Query plan

relation and one block from the inner relation. Depending on how much of the outer relation we fit in the buffer will determine how many times we will have to read the entire inner relation. **NOTE:** In this project, we will not consider using an index for the inner table access in the join algorithm. For the optimized query tree shown in Figure 4, The *Query Code Generator* produces the *Access Plan Table* as shown in Table 1. To get a better idea as to how the access plan was determined, let's examine step (3) in detail. For this example, let's assume the following: relation r2 has 1000 rows, each row requires 16 bytes of storage, the block size is 512 bytes. So, 32 rows will fit in a block, giving a total of 32 blocks for r2. The secondary index on attribute F allows for duplicate values, so buckets containing pointers to data blocks will be used in the index. A bucket contains only pointers (4 bytes) to data blocks and one pointer to an additional bucket. So, with a block size of 512, there will be 127 pointers in a bucket. Since we assume a uniform data distribution in our table and since attribute F has 50 values, each value will occur in 20 rows (i.e., $\frac{1000}{50}$). Hence, one bucket can store all the pointers to data blocks for a given value of F. So, using the secondary index for the selection operator ($F = 5$) will cost 1 I/O for the bucket access plus 20 I/Os for the data block accesses, giving a total of 23 I/Os. The alternative access method is a table scan

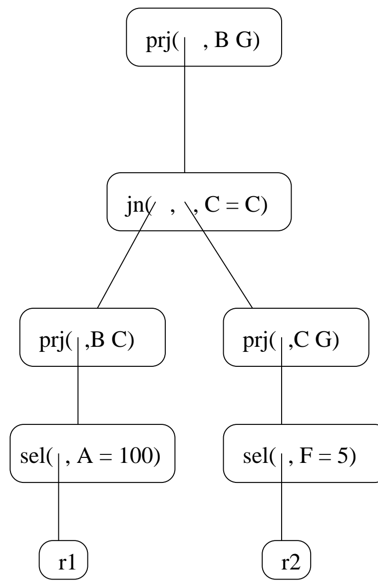

Figure 4. Another example of an optimized query tree

for relation $r2$, which would cost 32 I/Os for the data block accesses. So, using the index is more efficient.

Design of the fourth component

The *Runtime Database Processor* executes the functions for the statements in the access plan and prints the final result (i.e., table t6).