INCLUSION OF NEW TYPES IN RELATIONAL DATA BASE SYSTEMS

Michael Stonebraker
EECS Dept.
University of California, Berkeley

Abstract

This paper explores a mechanism to support user-defined data types for columns in a relational data base system. Previous work suggested how to support new operators and new data types. The contribution of this work is to suggest ways to allow query optimization on commands which include new data types and operators and ways to allow access methods to be used for new data types.

1. INTRODUCTION

The collection of built-in data types in a data base system (e.g. integer, floating point number, character string) and built-in operators (e.g. +, -, *, /) were motivated by the needs of business data processing applications. However, in many engineering applications this collection of types is not appropriate. For example, in a geographic application a user typically wants points, lines, line groups and polygons as basic data types and operators which include intersection, distance and containment. In scientific application, one requires complex numbers and time series with appropriate operators. In such applications one is currently required to simulate these data types and operators using the basic data types and operators provided by the DBMS at substantial inefficiency and complexity. Even in business applications, one sometimes needs user-defined data types. For example, one system [RTI84] has implemented a sophisticated date and time data type to add to its basic collection. This implementation allows subtraction of dates, and returns "correct" answers, e.g.

"April 15" - "March 15" = 31 days

This definition of subtraction is appropriate for most users; however, some applications require all months to have 30 days (e.g. programs which compute interest on bonds). Hence, they require a definition of sub-

---

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 83-0254 and the Naval Electronics Systems Command Contract N39-82-C-0235

traction which yields 30 days as the answer to the above computation. Only a user-defined data type facility allows such customization to occur.

Current data base systems implement hashing and B-trees as fast access paths for built-in data types. Some user-defined data types (e.g. date and time) can use existing access methods (if certain extensions are made); however other data types (e.g. polygons) require new access methods. For example R-trees [GUTM84], KDB trees [ROBI81] and Grid files are appropriate for spatial objects. In addition, the introduction of new access methods for conventional business applications (e.g. extendible hashing [FAGI79, LITW80]) would be expeditied by a facility to add new access methods.

A complete extended type system should allow:

1) the definition of user-defined data types
2) the definition of new operators for these data types
3) the implementation of new access methods for data types
4) optimized query processing for commands containing new data types and operators

The solution to requirements 1 and 2 was described in [STON83]; in this paper we present a complete proposal. In Section 2 we begin by presenting a motivating example of the need for new data types, and then briefly review our earlier proposal and comment on its implementation. Section 3 turns to the definition of new access methods and suggests mechanisms to allow the designer of a new data type to use access methods written for another data type and to implement his own access methods with as little work as possible. Then Section 4 concludes by showing how query optimization can be automatically performed in this extended environment.

2. ABSTRACT DATA TYPES

2.1. A Motivating Example

Consider a relation consisting of data on two dimensional boxes. If each box has an identifier, then it can be represented by the coordinates of two corner points as follows:

```
create box (id = i4, x1 = f8, x2 = f8, y1 = f8, y2 = f8)
```

Now consider a simple query to find all the boxes that overlap the unit square, ie. the box with coordinates (0, 1, 0, 1). The following is a compact representation of this request in QUEL:

```
retrieve (box.all) where not
  (box.x2 <= 0 or box.x1 >= 1 or box.y2 <= 0 or box.y1 >= 1)
```

The problems with this representation are:

The command is too hard to understand.

The command is too slow because the query planner will not be able to optimize something this complex.

The command is too slow because there are too many clauses to check.

The solution to these difficulties is to support a box data type whereby the box relation can be defined as:

```
create box (id = i4, desc = box)
```

and the resulting user query is:

```
retrieve (box.all) where box.desc !! "0, 1, 0, 1"
```

Here "!!" is an overlaps operator with two operands of data type box which returns a boolean. One would want a substantial collection of operators for user defined types. For example, Table 1 lists a collection of useful operators for the box data type.

Fast access paths must be supported for queries with qualifications utilizing new data types and operators. Consequently, current access methods must be extended to operate in this environment. For example, a reasonable collating sequence for boxes would be on ascending area, and a B-tree storage structure could be built for boxes using this sequence. Hence, queries such as

```
retrieve (box.all) where box.desc AE "0,5,0,5"
```

should use this index. Moreover, if a user wishes to optimize access for the !! operator, then an R-tree [GUTM84] may be a reasonable access path. Hence, it should be possible to add a user defined access method. Lastly, a user may submit a query to find all pairs of boxes which overlap, e.g:

```
range of b1 is box
range of b2 is box
retrieve (b1.all, b2.all) where b1.desc !! b2.desc
```

A query optimizer must be able to construct an access plan for solving queries which contains user defined operators.

Binary operator	symbol	left operand	right operand	result
overlaps	!!	box	box	boolean
contained in	<<	box	box	boolean
is to the left of	<L	box	box	boolean
is to the right of	>R	box	box	boolean
intersection	??	box	box	box
distance	"	box	box	float
area less than	AL	box	box	boolean
area equals	AE	box	box	boolean
area greater	AG	box	box	boolean

Unary operator	symbol	operand	result
area	AA	box	float
length	LL	box	float
height	HH	box	float
diagonal	DD	box	line

Operators for Boxes

Table 1

We turn now to a review of the prototype presented in [STON83] which supports some of the above function.

2.2. DEFINITION OF NEW TYPES

To define a new type, a user must follow a registration process which indicates the existence of the new type, gives the length of its internal representation and provides input and output conversion routines, e.g:

```
define type-name  length = value,
                  input  = file-name
                  output = file-name
```

The new data type must occupy a fixed amount of space, since only fixed length data is allowed by the built-in access methods in INGRES. Moreover, whenever new values are input from a program or output to a user, a conversion routine must be called. This routine must convert from character string to the new type and back. A data base system calls such routines for built-in data types (e.g. ascii-to-int, int-to-ascii) and they must be provided for user-defined data types. The input conversion routine must accept a pointer to a value of type character string and return a pointer to a value of the new data type. The output routine must perform the converse transformation.

Then, zero or more operators can be implemented for the new type. Each can be defined with the following syntax:

```
define operator  token = value,
                 left-operand = type-name,
                 right-operand = type-name,
                 result = type-name,
                 precedence-level like operator-2,
                 file = file-name
```

For example:

```
define operator  token = !!,
                 left-operand = box,
                 right-operand = box,
                 result = boolean,
                 precedence like *,
                 file = /usr/foobar
```

All fields are self explanatory except the precedence level which is required when several user defined operators are present and precedence must be established among them. The file /usr/foobar indicates the location of a procedure which can accept two operands of type box and return true if they overlap. This procedure is written in a general purpose programming language and is linked into the run-time system and called as appropriate during query processing.

2.3. Comments on the Prototype

The above constructs have been implemented in the University of California version of INGRES [STON76]. Modest changes were required to the parser and a dynamic loader was built to load the required user-defined routines on demand into the INGRES address space. The system was described in [ONG84].

Our initial experience with the system is that dynamic linking is not preferable to static linking. One problem is that initial loading of routines is slow. Also, the ADT routines must be loaded into data space to preserve sharability of the DBMS code segment. This capability requires the construction of a non-trivial loader. An "industrial strength" implementation might choose to specify the user types which an installation wants at the time the DBMS is installed. In this case, all routines could be linked into the run time system at system installation time by the linker provided by the operating system. Of course, a data base system implemented as a single server process with internal multitasking would not be subject to any code sharing difficulties, and a dynamic loading solution might be reconsidered.

An added difficulty with ADT routines is that they provide a serious safety loophole. For example, if an ADT routine has an error, it can easily crash the DBMS by overwriting DBMS data structures accidentally. More seriously, a malicious ADT routine can overwrite the entire data base with zeros. In addition, it is unclear whether such errors are due to bugs in the user routines or in the DBMS, and finger-pointing between the DBMS implementor and the ADT implementor is likely to result.

ADT routines can be run in a separate address space to solve both problems, but the performance penalty is severe. Every procedure call to an ADT operator must be turned into a round trip message to a separate address space. Alternately, the DBMS can interpret the ADT procedure and guarantee safety, but only by building a language processor into the run-time system and paying the performance penalty of interpretation. Lastly, hardware support for protected procedure calls (e.g. as in Multics) would also solve the problem.

However, on current hardware the prefered solution may be to provide two environments for ADT procedures. A protected environment would be provided for debugging purposes. When a user was confident that his routines worked correctly, he could install them in the unprotected DBMS. In this way, the DBMS implementor could refuse to be concerned unless a bug could be produced in the safe version.

We now turn to extending this environment to support new access methods.

3. NEW ACCESS METHODS

A DBMS should provide a wide variety of access methods, and it should be easy to add new ones. Hence, our goal in this section is to describe how users can add new access methods that will efficiently support user-defined data types. In the first subsection we indicate a registration process that allows implementors of new data types to use access methods written by others. Then, we turn to designing lower level DBMS interfaces so the access method designer has minimal work to perform. In this section we restrict our attention to access methods for a single key field. Support for composite keys is a straight forward extension. However, multidimensional access methods that allow efficient retrieval utilizing subsets of the collection of keys are beyond the scope of this paper.

3.1. Registration of a New Access Method

The basic idea which we exploit is that a properly implemented access method contains only a small number of procedures that define the characteristics of the access method. Such procedures can be replaced by others which operate on a different data type and allow the access method to "work" for the new type. For example, consider a B-tree and the following generic query:

retrieve (target-list) where relation.key OPR value

A B-tree supports fast access if OPR is one of the set:

{=, <, <=, >=, >}

and includes appropriate procedure calls to support these operators for a data type (s). For example, to search for the record matching a specific key value, one need only descend the B-tree at each level searching for the minimum key whose value exceeds or equals the indicated key. Only calls on the operator "<=" are required with a final call or calls to the routine supporting "=".

Moreover, this collection of operators has the following properties:

P1) key-1 < key-2 and key-2 < key-3 then key-1 < key-3
P2) key-1 < key-2 implies not key-2 < key-1
P3) key-1 < key-2 or key-2 < key-1 or key-1 = key-2
P4) key-1 <= key-2 if key-1 < key-2 or key-1 = key-2
P5) key-1 = key-2 implies key-2 = key-1
P6) key-1 > key-2 if key-2 < key-1
P7) key-1 >= key-2 if key-2 <= key-1

In theory, the procedures which implement these operators can be replaced by any collection of procedures for new operators that have these properties and the B-tree will "work" correctly. Lastly, the designer of a B-tree access method may disallow variable length keys. For example, if a binary search of index pages is performed, then only fixed length keys are possible. Information of this restriction must be available to a type designer who wishes to use the access method.

The above information must be recorded in a data structure called an access method **template**. We propose to store templates in two relations called TEMPLATE-1 and TEMPLATE-2 which would have the composition indicated in Table 2 for a B-tree access method. TEMPLATE-1 simply documents the conditions which must be true for the operators provided by the access method. It is included only to provide guidance to a human wishing to utilize the access method for a new data type and is not used internally in the system. TEMPLATE-2, on the other hand, provides necessary information on the data types of operators. The column "opt" indicates whether the operator is required or optional. A B-tree must have the operator "<=" to build the tree; however, the other operators are optional. Type1, type2 and result are possible types for the left operand, the right operand, and the result of a given operator. Values for these fields should come from the following collection;

a specific type, e.g. int, float, boolean, char
fixed, i.e. any type with fixed length
variable, i.e. any type with a prescribed varying length format
fix-var, i.e. fixed or variable
type1, i.e. the same type as type1
type2, i.e. the same as type2

After indicating the template for an access method, the designer can propose one or more collections of operators which satisfy the template in another relation, AM. In Table 3 we have shown an AM containing the original set of integer operators provided by the access method designer along with a collection

---

TEMPLATE-1	AM-name	condition
	B-tree	P1
	B-tree	P2
	B-tree	P3
	B-tree	P4
	B-tree	P5
	B-tree	P6
	B-tree	P7

TEMPLATE-2	AM-name	opr-name	opt	left	right	result
	B-tree	=	opt	fixed	type1	boolean
	B-tree	<	opt	fixed	type1	boolean
	B-tree	<=	req	fixed	type1	boolean
	B-tree	>	opt	fixed	type1	boolean
	B-tree	>=	opt	fixed	type1	boolean

Templates for Access Methods

Table 2

AM	class	AM-name	opr	generic name	opr-id opr	Ntups	Npages
	int-ops	B-tree	=	=	id1	N / Ituples	2
	int-ops	B-tree	<	<	id2	F1 * N	F1 * NUMpages
	int-ops	B-tree	<=	<=	id3	F1 * N	F1 * NUMpages
	int-ops	B-tree	>	>	id4	F2 * N	F2 * NUMpages
	int-ops	B-tree	>=	>=	id5	F2 * N	F2 * NUMpages
	area-op	B-tree	AE	=	id6	N / Ituples	3
	area-op	B-tree	AL	<	id7	F1 * N	F1 * NUMpages
	area-op	B-tree	AG	>	id8	F1 * N	F1 * NUMpages

The AM Relation

Table 3

added later by the designer of the box data type. Since operator names do not need to be unique, the field opr-id must be included to specify a unique identifier for a given operator. This field is present in a relation which contains the operator specific information discussed in Section 2. The fields, Ntups and Npages are query processing parameters which estimate the number of tuples which satisfy the qualification and the number of pages touched when running a query using the operator to compare a key field in a relation to a constant. Both are formulas which utilize the variables found in Table 4, and values reflect approximations to the computations found in [SELI79] for the case that each record set occupies an individual file. Moreover, F1 and F2 are surogates for the following quantities:

F1 = (value - low-key) / (high-key - low-key)
F2 = (high-key - value) / (high-key - low-key)

With these data structures in place, a user can simply modify relations to B-tree using any class of operators defined in the AM relation. The only addition to the modify command is a clause "using class" which specifies what operator class to use in building and accessing the relation. For example the command

```
modify box to B-tree on desc using area-op
```

will allow the DBMS to provide optimized access on data of type box using the operators {AE,AL,AG}. The same extension must be provided to the index command which constructs a secondary index on a field, e.g:

Variable	Meaning
N	number of tuples in a relation
NUMpages	number of pages of storage used by the relation
Ituples	number of index keys in an index
Ipages	number of pages in the index
value	the constant appearing in: rel-name.field-name OPR value
high-key	the maximum value in the key range if known
low-key	the minimum value in the key range if known

Variables for Computing Ntups and Npages

Table 4

---

index on box is box-index (desc) using area-op

To illustrate the generality of these constructs, the AM and TEMPLATE relations are shown in Tables 5 and 6 for both a hash and an R-tree access method. The R-tree is assumed to support three operators, contained-in (<<), equals (==) and contained-in-or-equals (<<=). Moreover, a fourth operator (UU) is required during page splits and finds the box which is the union of two other boxes. UU is needed solely for maintaining the R-tree data structure, and is not useful for search purposes. Similarly, a hash access method requires a hash function, H, which accepts a key as a left operand and an integer number of buckets as a right operand to produce a hash bucket as a result. Again, H cannot be used for searching purposes. For compactness, formulas for Ntups and Npages have been omitted from Table 6.

3.2. Implementing New Access Methods

In general an access method is simply a collection of procedure calls that retrieve and update records. A generic abstraction for an access method could be the following:

open (relation-name) — This procedure returns a pointer to a structure containing all relevant information about a relation. Such a "relation control block" will be called a descriptor. The effect is to make the relation accessible.

close (descriptor) — This procedure terminates access to the relation indicated by the descriptor.

TEMPLATE-1	AM-name	condition
	hash	Key-1 = Key-2 implies H(key1) = H(key-2)
	R-tree	Key-1 << Key-2 and Key-2 << Key-2 implies Key-1 << key-3
	R-tree	Key-1 << Key-2 implies not Key-2 << Key-1
	R-tree	Key-1 <<= Key-2 implies Key-1 << Key-2 or Key-1 == Key-2
	R-tree	Key-1 == Key-2 implies Key-2 == Key-1
	R-tree	Key-1 << Key-1 UU Key-2
	R-tree	Key-2 << Key-1 UU Key-2

TEMPLATE-2	AM-name	opr-name	opt	left	right	result
	hash	=	opt	fixed	type1	boolean
	hash	H	req	fixed	int	int
	R-tree	<<	req	fixed	type1	boolean
	R-tree	==	opt	fixed	type1	boolean
	R-tree	<<=	opt	fixed	type1	boolean
	R-tree	UU	req	fixed	type1	boolean

Templates for Access Methods

Table 5

AM	class	AM-name	opr name	generic opr	opr-id	Ntups	Npages
	box-ops	R-tree	==	==	id10		
	box-ops	R-tree	<<	<<	id11		
	box-ops	R-tree	<<=	<<=	id12		
	box-ops	R-tree	UU	UU	id13		
	hash-op	hash	=	=	id14		
	hash-op	hash	H	H	id15		

The AM Relation

Table 6

get-first (descriptor, OPR, value) — This procedure returns the first record which satisfies the qualification

..where key OPR value

get-next (descriptor, OPR, value, tuple-id) — This procedure gets the next tuple following the one indicated by tuple-id which satisfies the qualification.

get-unique (descriptor, tuple-id) — This procedure gets the tuple which corresponds to the indicated tuple identifier.

insert (descriptor, tuple)	This procedure inserts a tuple into the indicated relation
delete (descriptor, tuple-id)	This procedure deletes a tuple from the indicated relation.
replace (descriptor, tuple-id, new-tuple)	This procedure replaces the indicated tuple by a new one.
build (descriptor, keyname, OPR)	Of course it is possible to build a new access method for a relation by successively inserting tuples using the insert procedure. However, higher performance can usually be obtained by a bulk loading utility. Build is this utility and accepts a descriptor for a relation along with a key and operator to use in the build process.

There are many different (more or less similar) access method interfaces; see [ASTR76, ALLC80] for other proposals. Each DBMS implementation will choose their own collection of procedures and calling conventions.

If this interface is publicly available, then it is feasible to implement these procedures using a different organizing principle. A clean design of open and close should make these routines universally usable, so an implementor need only construct the remainder. Moreover, if the designer of a new access method chooses to utilize the same physical page layout as some existing access method, then replace and delete do not require modification, and additional effort is spared.

The hard problem is to have a new access method interface correctly to the transaction management code. (One commercial system found this function to present the most difficulties when a new access method was coded.) If a DBMS (or the underlying operating system) supports transactions by physically logging pages and executing one of the popular concurrency control algorithms for page size granules, (e.g. [BROW81, POPE81, SPEC83, STON85] then the designer of a new access method need not concern himself with transaction management. Higher level software will begin and end transactions, and the access method can freely read and write pages with a guarantee of atomicity and serializability. In this case the access method designer has no problems concerning transactions, and this is a significant advantage for transparent transactions. Unfortunately, much higher performance will typically result if a different approach is taken to both crash recovery and concurrency control. We now sketch roughly what this alternate interface might be.

With regard to crash recovery, most current systems have a variety of special case code to perform logical logging of events rather than physical logging of the changes of bits. There are at least two reasons for this method of logging. First, changes to the schema (e.g. create a relation) often require additional work besides changes to the system catalogs (e.g. creating an operating system file in which to put tuples of the relation). Undoing a create command because a transaction is aborted will require deletion of the newly created file. Physical backout cannot accomplish such extra function. Second, some data base updates are extremely inefficient when physically logged. For example, if a relation is modified from B-tree to hash, then the entire relation will be written to the log (perhaps more than once depending on the implementation of the modify utility). This costly extra I/O can be avoided by simply logging the command that is being performed. In the unlikely event that this event in the log must be undone or redone, then the modify utility can be rerun to make the changes anew. Of course, this sacrifices performance at recovery time for a compression of the log by several orders of magnitude.

If such logical logging is performed, then a new access method must become involved in logging process and a clean event-oriented interface to logging services should be provided. Hence, the log should be a collection of **events**, each having an event-id, an associated **event type** and an arbitrary collection of data. Lastly, for each event type, T, two procedures, REDO(T) and UNDO(T) are required which will be called when the log manager is rolling forward redoing log events and rolling backward undoing logged events respectively. The system must also provide a procedure,

LOG (event-type, event-data)

which will actually insert events into the log. Moreover, the system will provide a collection of **built-in event types**. For each such event, UNDO and REDO are available in system libraries. Built-in events would include:

replace a tuple
insert a tuple at a specific tuple identifier address
delete a tuple
change the storage structure of a relation
create a relation
destroy a relation

A designer of a new access method could use the built-in events if they were appropriate to his needs. Alternately, he could specify new event types by writing UNDO and REDO procedures for the events and making entries in a system relation holding event information. Such an interface is similar to the one provided by CICS [IBM80].

We turn now to discussing the concurrency control subsystem. If this service is provided transparently and automatically by an underlying module, then special case concurrency control for the system catalogs and index records will be impossible. This approach will severely impact performance as noted in [STON85]. Alternately, one can follow the standard scheduler model [BERN81] in which a module is callable by code in the access methods when a concurrency control decision must be made. The necessary calls are:

read (object-identifier)
write (object-identifier)
begin
abort
commit
savepoint

and the scheduler responds with yes, no or abort. The calls to begin, abort, commit and savepoint are made by higher level software, and the access methods need not be concerned with them. The access method need only make the appropriate calls on the scheduler when it reads or writes an object. The only burden which falls on the implementor is to choose the appropriate size for objects.

The above interface is appropriate for data records which are handled by a conventional algorithm guaranteeing serializability. To provide special case parallelism on index or system catalog records, an access method requires more control over concurrency decisions. For example, most B-tree implementations do not hold write locks on index pages which are split until the end of the transaction which performed the insert. It appears easiest to provide specific lock and unlock calls for such special situations, i.e:

lock (object, mode)
unlock (object)

These can be used by the access method designer to implement special case parallelism in his data structures.

The last interface of concern to the designer of an access method is the one to the buffer manager. One requires five procedures:

get (system-page-identifier)
fix (system-page-identifier)
unfix (system-page-identifier)
put (system-page-identifier)

order (system-page-identifier, event-id or system-page-identifier)

The first procedure accepts a page identifier and returns a pointer to the page in the buffer pool. The second and third procedures pin and unpin pages in the buffer pool. The last call specifies that the page holding the given event should be written to disk prior to the indicated data page. This information is necessary in write-ahead log protocols. More generally, it allows two data pages to be forced out of memory in a specific order.

An access method implementor must code the necessary access method procedures utilizing the above interfaces to the log manager, the concurrency control manager and the buffer manager. Then, he simply registers his access method in the two TEMPLATE relations.

3.3. Discussion

A transparent interface to the transaction system is clearly much preferred to the complex collection of routines discussed above. Moreover, the access method designer who utilizes these routines must design his own events, specify any special purpose concurrency control in his data structures, and indicate any necessary order in forcing pages out of the buffer pool. An open research question is the design of a simpler interface to these services that will provide the required functions.

In addition, the performance of the crash recovery facility will be inferior to the recovery facilities in a conventional system. In current transaction managers, changes to indexes are typically not logged. Rather, index changes are recreated from the corresponding update to the data record. Hence, if there are n indexes for a given object, a single log entry for the data update will result in n+1 events (the data update and n index updates) being undone or redone in a conventional system. Using our proposed interface all n+1 events will appear in the log, and efficiency will be sacrificed.

The access method designer has the least work to perform if he uses the same page layout as one of the built-in access methods. Such an access method requires get-first, get-next, and insert to be coded specially. Moreover, no extra event types are required, since the built-in ones provide all the required functions. R-trees are an example of such an access method. On the other hand, access methods which do not use the same page layout will require the designer to write considerably more code.

4. QUERY PROCESSING AND ACCESS PATH SELECTION

To allow optimization of a query plan that contains new operators and types, only four additional pieces of information are required when defining an operator. First, a selectivity factor, Stups, is required which estimates the expected number of records satisfying the clause:

...where rel-name.field-name OPR value

A second selectivity factor, S, is the expected number of records which satisfy the clause

...where relname-1.field-1 OPR relname-2.field-2

Stups and S are arithmetic formulas containing the predefined variables indicated earlier in Table 4. Moreover, each variable can have a suffix of 1 or 2 to specify the left or right operand respectively.

Notice that the same selectivity appears both in the definition of an operator (Stups) and in the entry (Ntups) in AM if the operator is used in an index. In this case, Ntups from AM should be used first, and supports an if-then-else specification used for example in the [SELI79] for the operator "=" as follows:

selectivity = (1 / Ituples) ELSE 1/10

In this example selectivity is the reciprocal of the number of index tuples if an index exists else it is 1/10. The entry for Ntups in AM would be (N / Ituples) while Stups in the operator definition would be N / 10.

The third piece of necessary information is whether merge-sort is feasible for the operator being defined. More exactly, the existence of a second operator, OPR-2 is required such that OPR and OPR-2 have properties P1-P3 from Section 3 with OPR replacing "=" and OPR-2 replacing "<". If so, the relations to be joined using OPR can be sorted using OPR-2 and then merged to produce the required answer.

The last piece of needed information is whether hash-join is a feasible joining strategy for this operator. More exactly, the hash condition from Table 6 must be true with OPR replacing "=".

An example of these pieces of information for the operator, AE, would be:

```
define operator  token = AE,
                 left-operand = box,
                 right-operand = box,
                 result = boolean,
                 precedence like *,
                 file = /usr/foobar,
                 Stups = 1,
                 S = min (N1, N2),
```

merge-sort with AL,
hash-join

We now turn to generating the query processing plan. We assume that relations are stored keyed on one field in a single file and that secondary indexes can exist for other fields. Moreover, queries involving a single relation can be processed with a scan of the relation, a scan of a portion of the primary index, or a scan of a portion of one secondary index. Joins can be processed by iterative substitution, merge-sort or a hash-join algorithm. Modification to the following rules for different environments appears straigth-forward.

Legal query processing plans are described by the following statements.

1) Merge sort is feasible for a clause of the form:

relname-1.field-1 OPR relname-2.field-2

if field-1 and field-2 are of the same data type and OPR has the merge-sort property. Moreover, the expected size of the result is S. The cost to sort one or both relations is a built-in computation.

2) Iterative substitution is always feasible to perform the join specified by a clause of the form:

relname-1.field-1 OPR relname-2.field-2

The expected size of the result is calculated as above. The cost of this operation is the cardinality of the outer relation multiplied by the expected cost of the one-variable query on the inner relation.

3) A hash join algorithm can be used to perform a join specified by:

relname-1.field-1 OPR relname-2.field-2

if OPR has the hash-join property. The expected size of the result is as above, and the cost to hash one or both relations is another built-in computation.

4) An access method, A for relname can be used to restrict a clause of the form

relname.field-name OPR value

only if relname uses field-name as a key and OPR appears in the class used in the modify command to organize relname. The expected number of page and tuple accesses are given by the appropriate row in AM.

5) A secondary index, I for relname can be used to restrict a clause of the form:

relname.field-name OPR value

only if the index uses field-name as a key and OPR appears in the class used to build the index. The expected number of index page and tuple accesses is given by the appropriate row in AM. To these must be added 1 data page and 1 data tuple per index tuple.

6) A sequential search can always be used to restrict a relation on a clause of the form:

relname.field-name OPR value

One must read NUMpages to access the relation and the expected size of the result is given by Stups from the definition of OPR.

A query planner, such as the one discussed in [SELI79] can now be easily modified to compute a best plan using the above rules to generate legal plans and the above selectivities rather than the current hard-wired collection of rules and selectivities. Moreover, a more sophisticated optimizer which uses statistics (e.g. [KOOI82, PIAT84] can be easily built that uses the above information.

5. CONCLUSIONS

This paper has described how an abstract data type facility can be extended to support automatic generation of optimized query processing plans, utilization of existing access methods for new data types, and coding of new access methods. Only the last capability will be difficult to use, and a cleaner high performance interface to the transaction manager would be highly desirable. Moreover, additional rules in the query optimizer would probably be a useful direction for evolution. These could include when to cease investigating alternate plans, and the ability to specify one's own optimizer parameters, e.g. the constant W relating the cost of I/O to the cost of CPU activity in [SELI79].

REFERENCES

[ALLC80] Allchin, J. et. al., "FLASH: A Language Independent Portable File Access Method," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[ASTR76] Astrahan, M. et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

[BERN81] Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, June 1981.

[BROW81] Brown, M. et. al., "The Cedar DBMS: A Preliminary Report," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.

[FAGI79] Fagin, R. et. al., "Extendible Hashing: A Fast Access Method for Dynamic Files," ACM-TODS, Sept. 1979.

[GUTM84] Gutman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[IBM80] IBM Corp, "CICS System Programmers Guide," IBM Corp., White Plains, N.Y., June 1980.

[KOOI82] Kooi, R. and Frankfurth, D., "Query Optimization in INGRES," IEEE Database Engineering, September 1982.

[LITW80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," Proc. 1980 VLDB Conference, Montreal, Canada, October 1980.

[ONG84] Ong, J. et. al., "Implementation of Data Abstraction in the Relational System, INGRES," ACM SIGMOD Record, March 1984.

[PIAT84] Piatetsky-Shapiro, G. and Connell, C., "Accurate Estimation of the Number of Tuples Satisfying a Condition," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[POPE81] Popek, G., et. al., "LOCUS: A Network Transparent, High Reliability Distributed System," Proc. Eighth Symposium on Operating System Principles, Pacific Grove, Ca., Dec. 1981.

[RTI84] Relational Technology, Inc., "INGRES Reference Manual, Version 3.0," November 1984.

[ROBI81] Robinson, J., "The K-D-B Tree: A Search Structure for Large Multidimensional Indexes," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.

[SELI79] Selinger, P. et. al., "Access Path Selection in a Relational Database Management System," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[SPEC83] Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing," Operating Systems Review, Vol 17, No 2, April 1983.

[STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON83] Stonebraker, M. et. al., "Application of Abstract Data Types and Abstract Indices to CAD Data," Proc. Engineering Applications Stream of Database Week/83, San Jose, Ca., May 1983.

[STON85] Stonebraker, M. et. al., "Interfacing a Relational Data Base System to an Operating System Transaction Manager," SIGOPS Review, January 1985.