Starburst Mid-Flight: As the Dust Clears

LAURA M. HAAS, WALTER CHANG, GUY M. LOHMAN, JOHN McPHERSON, MEMBER, IEEE, PAUL F. WILMS, GEORGE LAPIS, BRUCE LINDSAY, HAMID PIRAHESH, MICHAEL J. CAREY, AND EUGENE SHEKITA

Abstract—**The Starburst project, at IBM's Almaden Research Center, is improving the design of relational database management systems and enhancing their performance, while building an *extensible* system to better support nontraditional applications (such as engineering, geographic, office, etc.), and to serve as a testbed for future improvements in database technology. As of November 1989, we have an initial prototype of our system up and running. In this paper, we reflect on the design and implementation of the Starburst system to date. We examine some key design decisions, and how they affect the goal of improved structure and performance. We also examine how well we have met our goal of extensibility: what aspects of the system are extensible, how extensions can be done, and how easy it is to add extensions. We discuss some actual extensions to the system, including the experiences of our first real customizers.**

Index Terms—**Access methods, data structures, extensibility, plan optimization, query processing, relational database system, rule systems.**

I. INTRODUCTION

A. Starburst Goals

THE goal of the Starburst project at IBM's Almaden Research Center is to build a relational database management system that can be easily extended to support applications and technologies not typically supported by today's relational systems, and can serve as a malleable testbed to pursue our diverse research interests. In addition, we want to reexamine traditional system structures and algorithms, with the goal of improving their performance.

These goals are motivated by several observations. The broad acceptance of relational databases demonstrates their usefulness for classical applications. Traditional relational systems, however, do not provide adequate support for applications such as CAD/CAM, office systems, statistical databases, expert systems, and text applications because these systems lack the data types, functions, complex object support, storage techniques, and access methods that these applications require for ease of use and performance. We thus felt that the best way to provide database support for these new applications while continuing to provide support for traditional applications was to build a relational database that can be easily extended, and we set for ourselves the very broad goal of building a complete database system including concurrency control, recovery, authorization, language processing, and optimization, and to explore extensibility in every aspect of database management. We purposely avoided concentrating on one or two applications, because we did not want a narrow focus to inadvertently constrain the scope of extensibility in Starburst. While extensibility is an important goal, it should not be obtained at the expense of performance. Thus, we chose as a second goal for Starburst to take some of the best ideas for building high-performance systems, to improve on them where possible, and to incorporate them into the structure and algorithms of Starburst.

In its breadth and completeness, we feel that Starburst goes beyond other extensible systems, including AIM [14], DASDBS [42], Exodus [7], Genesis [4], Postgres [47], PROBE [18], RAD [40], and SABRE [1]. We also feel that our goal is more ambitious than object-oriented systems (e.g. ORION [3], O2 [2]) because we want to accommodate not only objects, but also to be able to adapt to new technologies and extensions based on a set-oriented, declarative language. The breadth and completeness of Starburst has provided us with many challenges, but we feel we have learned many valuable lessons. This paper will describe some of these, both in terms of what we feel we did correctly, and what we feel we would do differently if we were to do things over again.

Starburst can be viewed as having two phases: the first phase is the implementation of an extensible relational database management system with improved structure and performance; the second phase will be to use and extend Starburst as a testbed for our diverse research interests, including user-defined datatypes, complex objects, support for rules, main-memory databases, and parallelism. At this point we have made substantial progress on the first phase and have begun work on the second phase. We have an initial prototype of our system up and running. As of November 1989, we are able to compile and execute single table queries using relation scans and indexes, and multitable statements using nested block joins. Some advanced features of our language, e.g., table functions, are also supported. Individual components support a great deal more function than that, and should be integrated fairly quickly. In particular, additional join methods and

Manuscript received August 1, 1989; revised December 4, 1989.

L. M. Haas, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, and H. Pirahesh are with IBM Almaden Research Center, San Jose, CA 95120.

W. Chang is with Oracle Corporation, Belmont, CA 94002.

M. J. Carey and E. Shekita are with the Department of Computer Science, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8933787.

recursive queries should be running soon. Meanwhile, the system is sufficiently operational for us to experiment with it, and we have already built and tested several extensions.

In this paper, we reflect on the design and implementation of the Starburst system to date. In the next section, we will look at some key design decisions, and how they affect the goal of improved structure and performance. In Section III, we examine what aspects of the system are extensible and how extensions are done. Section IV will describe actual extensions to the system, including the experiences of our first real customizers. Section V looks at the implementation itself, and some lessons we have learned the hard way. We give some preliminary performance figures in Section VI, and suggest some future directions for research in the Section VII. Due to the nature of this special issue, we have tried to avoid a full description of the Starburst system. Companion papers will be referenced throughout the text that provide more detailed descriptions of various Starburst components. However, a brief overview is provided here to make the rest of the paper intelligible.

B. Starburst Overview

Starburst is designed to be a fully functional relational database management system, which can be *customized* (extended) by highly skilled programmers with expertise in database systems. We call these programmers (*database*) *customizers*. Starburst consists of a query language processor, Corona [24], and a data manager, Core. Core and Corona correspond roughly to System R's RSS and RDS, respectively. Corona compiles queries in an extended version of the SQL language, called *Hydrogen* (what else?), into calls on the underlying Core services to fetch and modify data.

As shown in Fig. 1, there are five main phases in Corona: parsing, query rewrite optimization, plan optimization, query refinement, and query evaluation (QES). The first four phases are part of query compilation, and the last phase is the run-time portion of Corona. Corona builds and uses two main data structures: the *query graph model* (*QGM*), an internal semantic network that describes the query during compilation, and the *query evaluation plan* (*QEP*, or *plan*), which is the output of the compilation phase and is used at run time to execute the query. The Parser creates a QGM from the incoming Hydrogen query. During query rewrite [25], the QGM is transformed into an equivalent QGM using context-dependent rules written in C.[1] The current transformations include merging views, converting subqueries to joins (when possible), and applying selection predicates, column projections, and elimination of duplicates as soon as possible. The goal of query rewrite is to create a QGM which will lead to a better execution strategy when processed by plan optimization. When a transformation is not clearly beneficial, we plan to retain both the original and transformed representations

[1]C is a trademark of AT&T

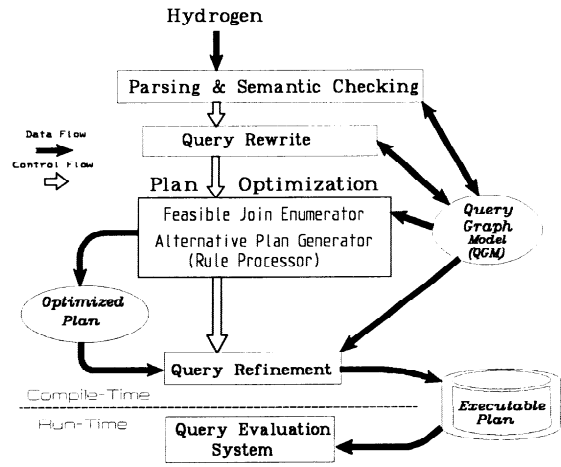


Fig. 1. Corona.

and connect them by a CHOOSE operator [25], [22], allowing plan optimization to evaluate both.[2] Plan optimization evaluates possible execution strategies for a given QGM based on a model of execution costs. The Starburst plan optimizer consists of a join enumerator, which determines feasible join orderings [39], and a plan generator, which constructs execution plans for those joins using a set of context-free, grammar-like production rules [35], [30]. Query refinement repackages the optimized plan for more efficient execution. The query evaluation system executes a QEP against the database. Our QES is similar to those of Exodus and Genesis [7], [4]. Each QES routine interprets one QEP operator. Each operator takes one or more streams of tuples as input and produces one or more streams of tuples as output.

Core stores and retrieves data, maintains access paths to the data, and ensures transaction consistency and data integrity in the face of multiple users and failures. One of the key features of Core is the data management extension architecture [32] shown in Fig. 2. This architecture allows two important types of extensions to Starburst called *storage methods* and *attachments*. Storage methods provide alternative methods for implementing tables, and can be tailored to fit the needs of various applications and technologies. A default storage method is provided that supports recoverable tables, but a user of Starburst is able to specify at data definition time that any other implemented storage method be used when creating a table. Attachments provide access paths, integrity constraints, and trigger extensions for tables. Storage method update, insert, and delete operations are called directly by Corona, and the corresponding attachment calls are invoked automatically by Core and thus indirectly by Corona. This ensures that consistency is maintained between storage methods and attachments. Core uses a *Core relation descriptor* to determine the attachments that must be called automatically, and locates the correct storage method or attachment routines to call for a particular operation through the use of different vectors of storage method or

[2]A better integration of query rewrite and plan optimization is needed. At the moment, we have no idea how to achieve this.

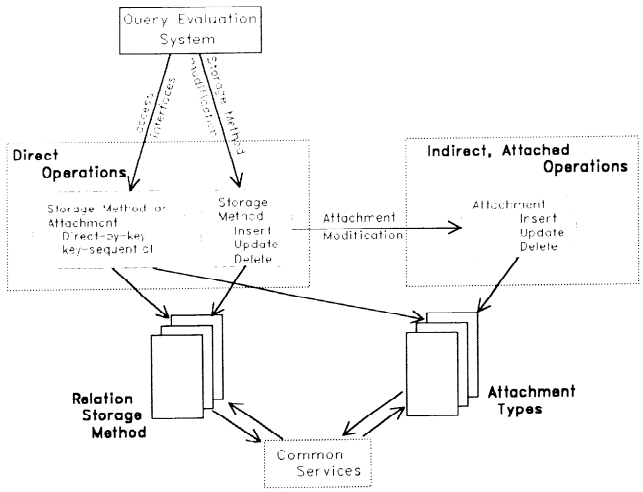


Fig. 2. Core data management extension architecture.

attachment functions. Scan and fetch operations on both storage methods and attachments are called directly by Corona. Storage methods and attachments can invoke any Core system service, including logging, recovery management, a predicate evaluator, event queues, a lock manager, interfaces to operating system services, debugging and tracing, and error reporting services.

II. Improved Internal Structure

One of the greatest benefits of starting a new system from scratch is the ability to make major improvements in internal structure that would be difficult to experiment with in an existing system. The gain is enhanced significantly if at least some of the people working on the new system have some experience doing the same or similar things. This proved especially true for Starburst. Almost half of the project members had come from the previous R* prototype distributed relational DBMS [51], [34]. Yet there had been enough turnover from the original builders of System R that few R* people were willing to defend System R approaches based upon loyalty or "pride of authorship" alone. The result was a design for Starburst that incorporated many substantial improvements suggested by a decade of sometimes difficult experience. Of course, improvements in structure do not always advance the cause of high performance.

Our enhancements to the internal structure of Starburst were of three types: 1) richer data structures that better matched their ultimate use, 2) better integration and interchangeable use of components, and 3) just plain careful engineering. We will illustrate each of these themes with several examples from Starburst.

A. Richer Internal Representations

One of the strongest motivations for starting Starburst from scratch was the need for revising the major internal data structures to capture richer semantics. Such structures usually streamline the code that uses them, but complicate the code that creates and maintains them. The new Starburst data structures, described below, were unqualified successes, despite their complexity.

1) Query Graph Model (QGM): In our earlier systems, the parse tree was used throughout query processing as *the* authoritative description of the query, even though additional (and often redundant) data structures were built for major query entities such as tables, columns, and predicates. Unfortunately, parse trees are designed to facilitate parsing and little else; the inevitable forays into the parse tree later in query processing became programming nightmares of abstruse code, redundant data structures, and complex bookkeeping.

The solution in Starburst was to create during parsing a semantic network describing the query, for subsequent use by all query processing. Inspired by the entity-relationship model, QGM maintains attributes of query entities such as tables, predicates, and columns, as well as the relationships between them, such as in which table each column appears. Considerable effort was expended in the design of this internal database to ensure that the entities and relationships modeled were essential yet not exclusive, so that new ones could be added relatively easily without affecting existing relationships. In addition, the design emphasized *consistency* and *orthogonality*, allowing us to remove some ad hoc restrictions on the semantics of SQL that had manifested themselves as rather arbitrary-looking limitations on the syntax of SQL. By providing a compact, detailed, understandable, and unambiguous representation of a query, QGM greatly simplified our internal language processing. A complete description of QGM, and the powerful query transformations it permits, can be found in [25].

2) Plans as Operator Trees: One of the least extensible aspects of System R was the structure of its query evaluation plans. Plans were *operand*-oriented, a sequence of *miniplans*, one miniplan per table accessed. Information on *operations*, such as join methods and sorts, was embedded within each miniplan. As the number of possible operations increased, the miniplan structure became increasingly complex and sparse. The shortcomings of this structure became even more apparent with the additional operations needed in R*, for which special "pseudo-miniplans" had to be devised [15], [16], [33]. Because of the operand orientation, the modules that translated the QEP into an executable sequence of low-level operations [36] became increasingly complex.

In Starburst, all of these problems were eliminated and extensibility enhanced by orienting plans toward an extensible set of *operators* (rather than *operands*) ([45], [53], or references in [28]). Operators in Starburst's QEP's are called *Low-Level Plan Operators*, or LOLEPOPs [35].

LOLEPOPs and the operator-oriented structure of QEP's simplify query processing. Most importantly, translating the plan chosen by the plan optimizer into a plan interpretable by the query evaluation system (QES) is simplified considerably, because the LOLEPOPs produced by the plan optimizer are precisely those that QES

can interpret. Query refinement only needs to convert operands of the LOLEPOPs from pointers to internal (QGM) data structures into pointers to externally referenceable table descriptors that will be stored with the plan. Second, LOLEPOPs provide a convenient way to modularize the plan generator functions (called *property functions*) that determine the properties of a plan, including its estimated execution cost. A property function effectively summarizes the impact that a LOLEPOP has on its input tables. In this way, new sequences of existing LOLEPOPs do not require the definition of new cost estimates; the cost equations for the LOLEPOPs are simply composed in a new way.

3) Nonnormalized Meta-Data: Meta-data describing the database is usually organized into two distinct levels. A higher level describes the external logical and physical schema. This information is used for semantic query analysis and for query optimization. Lower level meta-data (discussed later) are used during query execution by the data management layer to physically access and manipulate the data.

One of the touted virtues of relational systems is that the high-level meta-data are stored in tables and can be queried just like regular data. However, the cost of accessing this information during query compilation is then proportional to the number of columns and access paths (e.g., indexes) defined for the table. For simple queries, this cost can dominate query compilation time, since a record fetch (and hence a disk I/O, unless the records are clustered) is needed for each column or index of each table of the query. (Note that many "real" tables may have several dozen columns.)

Starburst reduces the cost of accessing a table's high-level meta-data, called the *Corona descriptor*, by denormalizing the "system" catalogs into three records per table: one record for the table and its attachments (table name, cardinality, storage method, and attachment descriptions), one record for column names, and one record for column characteristics (type, length, cardinality, etc.). Thus, the cost of accessing the data needed to analyze and optimize a query is proportional to the number of tables, not columns, a substantial reduction. We plan to maintain a second set of fully normalized "user" catalogs, with the same information, for access via schema browsers and user queries.

B. Integration and Interchangeability of Components

A second major motivation for starting from scratch in Starburst was the large gulf separating the RSS from the RDS. We resolved to better integrate Corona and Core, using uniform internal interfaces wherever possible to reduce conversions, and providing "big picture" information that could enhance the performance of the data manager. These changes, some of which are described below, also paid big dividends, but not without some costs.

1) Uniform Record Structure: In Starburst, one uniform record representation is used throughout the system, to provide a self-contained and uniform interface for many components. Each record consists of a contiguous stream of bytes structured into a header, offset directory, and data area. A common set of primitives provides all of the operations needed to create and modify these records, thus simplifying record manipulations. Each Starburst record can support a variable number of fields, each of varying length. If necessary, arbitrary nesting of other records can occur by storing an entire new record within a field. As it turned out, this nesting feature was used quite heavily in our internal implementation to manage various catalog descriptors (see Section II-A3).

The advantages of our record approach are uniform treatment of nulls and varying-length fields, uniform access to any field in the record, and a standard set of primitives for storage methods and attachments to manipulate all data internally. However, if we were to do it over, it is not clear that we would use this format. The record structure has had a negative impact on performance for fixed length records. This is because the size of the directory and the amount of padding needed for alignment reduce the number of records we can fit on a page. For example, for the Wisconsin Benchmarks [6], Starburst gets 15 records to a page, as compared to 20 per page for DB2. Thus, the number of I/O's we do is increased substantially. However, prefetching pages might hide some of this problem.

Furthermore, the alignment of data within fields, as required by our workstation hardware, complicates the manipulation of the record structures considerably. We have not yet found an entirely satisfactory solution to this problem. If we were to use this structure again, we would support a larger number of fields (our current limit is 255 fields) and add more space in the record headers to characterize the record.

2) Expression Evaluation: Systems R and R* had two separate predicate evaluators in their two major components: 1) a very simple type-insensitive evaluator in the RSS that used lexical comparison to evaluate predicates of the form "<column> <relational-operator> <value>," and 2) an evaluator "of last resort" in the RDS that could evaluate complex expressions of mixed types. The RSS evaluator could apply its limited set of predicates very efficiently before a tuple was extracted from its page in the buffer. The RDS evaluator, on the other hand, operated only on tuples extracted from the page and returned by the RSS to the RDS.

In Starburst, we sought to unify and thus simplify all expression evaluation—including select-list expressions as well as predicate expressions—in a single evaluator that could be invoked anywhere and applied to any data. The uniform record structure allowed us to satisfy the latter requirement. At compile time, the expression evaluator translates a parse tree for an expression into a stack-machine program for the run-time part of the expression evaluator. Expressions are evaluated uniformly on records in buffer pages, previously fetched fields, host variables, and query constants. We expect performance to be roughly comparable to earlier systems: a bit worse on very

simple predicates due to the stack architecture, but better on more complex predicates because we can apply them earlier.

3) Replacement Hints to the Buffer Pool Manager: The buffer pool manager of System R used a standard least recently used (LRU) protocol for choosing a page to replace. In many cases, however, the plan chosen by the optimizer can provide much better information about the likelihood of reusing a given page [11]. For example, the pages of the outermost table of a multitable query will be read at most once and never be revisited again by that query. On the other hand, the innermost pages are very likely to be revisited many times, so we would like to keep them in the buffer if possible, even though we may access many other pages in the meantime.

Starburst's buffer pool manager (BPM) uses the clock algorithm [44] (an approximation to LRU), which provides a mechanism for execution strategies, storage methods, and attachments to give hints to BPM about expected future use of a page. BPM will attempt to favor or disfavor the page accordingly when making page replacement decisions. So, in our example, outermost pages will be disfavored and innermost pages will be favored. Using this mechanism, BPM's replacement algorithm can be tuned to use the buffer pool more efficiently for new extensions in query processing and the data manager.

The BPM hint mechanism was used extensively in an implementation of the Grace hash join method [29]. The mechanism was shown to be powerful enough to permit the hash join to control the number of buffers used, and to control their contents for optimal reuse of buffered data [31]. These results were verified in a single-user environment, but more work needs to be done to assess the effects of concurrent transactions.

4) Cached Meta-Data: In System R, the RSS is exclusively responsible for managing the low-level meta-data for a table, which it uses at run-time to specify the physical location and characteristics of stored data. The RDS passes "handles" for this meta-data to the RSS, which must then locate and retrieve its meta-data from permanent storage, incurring the cost of buffer pool accesses and possibly some I/O.

In Starburst, the low-level meta-data (Core descriptor) are cached within the table record of the Corona descriptor. The query compiler copies the Core descriptor into the query plan and, at run time, a direct pointer to the descriptor is passed to Core. This improves run-time performance significantly. However, it also increases the size of query plans, and will require invalidation of query plans whenever the Core descriptors are changed in any way. In contrast, System R required less frequent invalidation; for example, plans were invalidated whenever an index was dropped, if the plan depended on that index, but not when an index was added.

C. Careful Engineering

Careful engineering with performance in mind yields big gains, both in simplified code and in better performance. Often the benefits are compounded in other, seemingly unrelated components. We give three examples from Starburst below.

1) Single-Pass Parser: System R used a standard parser that completely parsed an SQL statement in the order in which it was entered, with the SELECT clause first. Once the SQL statement was parsed, catalog information for the tables referenced in the table list could be retrieved from disk, including the columns that those tables contained. When one of the tables was a view, its definition would also be retrieved and merged with the original query. Only then could the validity of columns referenced in the SELECT clause and the predicates be verified in a second "semantic check" pass over the parse tree, long after pointers into the original text of the SQL statement had been lost. The result was inefficient processing, and often cryptic syntax error messages.

If only SQL listed the tables first, as is done, for example, in the RANGE statement in QUEL [47]! This observation suggested a clever solution: while lexically scanning the SQL statement, move the SELECT clause after the WHERE clause, leaving the FROM clause the first to be parsed. Now as each table or view in the FROM clause is parsed, we can look up its catalog information and construct its QGM information. Column references in the WHERE and SELECT clauses that follow can then be verified against QGM during parsing, while syntactic context and pointers to the input string are available and meaningful. The definition of any view in the FROM clause is parsed recursively, and its QGM merged with that of the query. Best of all, this approach requires only a single pass through the SQL statement, streamlining the code significantly!

2) Write-Ahead Logging: System R used shadow pages and a log to provide recovery from system crashes and to handle partial transaction rollback during normal processing. The shadowing technique allowed the system to start recovery in a physically consistent state, but it consumed large amounts of disk space, destroyed data clustering, and resulted in unpredictable response times because of the need to quiesce RSS activity while taking a checkpoint to update the backup copy. DB2 [13] uses write-ahead logging (WAL), but its WAL scheme does not support fine granularity locking, such as record locking.

Starburst implements a write-ahead log protocol [38] that has many advantages over previous methods. It can support any locking granularity, including record-level locking, so that storage methods and attachments can use the granularity of locking that is most appropriate. For example, the default storage method supports record-level locking, and the *B*-tree index attachment locks partitions of keys on a page, instead of entire pages, for increased concurrency. The Starburst WAL protocol also consumes less log space than the DB2 method, reduces the checkpointing overhead of the System R approach, and preserves data clustering.

3) Compression of Keys in B-Trees: In System R, each leaf node of a *B*+ tree index contained a key value, fol-

lowed by an ordered list of TID's of records having that key value, providing some compression of duplicate key values. In Starburst, keys are represented using the uniform record structure previously described, and are stored on fixed-size 4K byte leaf pages in the $B+$ tree. Two different types of key compression are used. In each node or leaf page, successive keys are front (prefix) compressed. When significant front compression occurs, page I/O's during index scans are reduced, since fewer leaf pages need to be accessed. We have observed front compression rates of 20–35% per page on actual data taken from the on-line IBM telephone directory. Front compression also allows us to avoid special logic for efficient duplicate key storage, but usually hampers the efficiency of binary searching. We have devised a new prefix compression method that still permits efficient binary searching [10].

We also use rear compression in our $B+$ trees. In each interior page, each dividing key forms an upper bound value for its lower level child page and is rear (suffix) compressed. The rear compression removes any suffix portion of a dividing key that does not contribute information to the $B+$ tree search decision when a root-down search is being performed. We have observed rear compression rates of 15–25% per interior page, again on the on-line telephone directory. The net effect of rear compression is to increase the density of dividing keys on interior pages, thereby increasing the fan-out to lower level pages. This delays the need for higher level page splits and reduces the tree's height, thus reducing the disk I/O's to access the tree.

While compression strategies introduce more complexity during inserts, in our view the space and I/O savings justify the use of both types of compression. IBM's VSAM [26], [27] has used both types of compression for years [49], [12], [8]. If minimum complexity were a constraint, we would still advocate keeping rear compression, since the reduced I/O can be obtained at very little cost in terms of additional complexity. This is not true for front compression in our scheme.

III. Extensibility

In Starburst, we set ourselves the goal of being sufficiently extensible to accommodate any new technology for any component and to support efficiently any application. We have made a fair amount of progress towards this ambitious goal. Starburst is *very* extensible. It can accommodate many new technologies, though probably not all. We have not yet tested it with a complete implementation of an extension for a specific application.

Virtually all components of Starburst are extensible. This is important because to support even simple extensions, changes in many components may be needed. For example, to add a new operation, we must extend the query language, and be able to represent the changes internally (in QGM). For efficiency and correctness, we must specify what semantic optimizations apply. The plan optimizer must be able to use the operation in a plan, and the evaluation system must be able to execute it. We may want a special privilege to control the set of users who may execute the new operation. Likewise, if we add a new attachment, we must modify the data definition language so that users can define instances of the new attachment. We may also want to alter the recovery, locking, and other support services for the new attachment. Applications will require still other forms of extensibility from the system, including support for new datatypes and objects, new orderings of data, and user-defined rules.

In this section, we examine in what ways Starburst is extensible, what problems are introduced by this extensibility, what aspects should be more extensible, and how we achieved the level of extensibility we have. In the next section, we describe several extensions to Starburst, and what we learned from them.

A. Extensibility Throughout Starburst

1) Supporting New Operations:

Increasing Expressiveness: Starburst's query language, Hydrogen [24], offers the customizer two powerful ways of adding new capabilities. *Table expressions* [17] named queries that may appear wherever a table would be allowed, have already enabled us to add support for recursion, and permit complex computations to be expressed. Four types of *functions* can be defined, ranging from simple scalar functions (e.g., *sin(angle)*) on one or more columns to *table functions*, which produce a table from zero or more input tables. Table functions can be particularly useful for defining new operations (e.g., relational divide, or *sample(table, int)* for statistical sampling).

Hydrogen has a relatively small number of built-in constructs. This keeps the grammar small and compact, and perhaps makes the language easier to learn. However, it creates two difficulties. First, while complex queries can be expressed, their representation in Hydrogen is usually quite complex as well. This could make Hydrogen very hard to read and write, making application development difficult. In addition, even with a small number of constructs, there are frequently several ways of expressing the same query. Whenever feasible, the performance of a query should depend on its *meaning* rather than on its *expression*; this increases nonprocedurality. Identifying equivalent expressions for a query poses a significant challenge for optimization. We do not have sufficient experience with applications or performance to judge the extent of these potential difficulties, but we hope that the use of views will ease the first, and that our query rewrite optimizer will mitigate the second.

Representing Additional Semantics: Almost any change in the functionality of Hydrogen impacts the internal representation of that functionality, in QGM. We anticipate that the most common extensions will be to add new operations on tables and to enhance semantic processing. Since QGM is exposed to the customizer, virtually any extension is possible. In most cases, extensions require only minor changes to QGM, because much of QGM is "generic;" that is, the information is standard and has a

standard interpretation, regardless of the operation. This is because most of QGM describes *tables*, both input and output, and not the operations on tables.

As QGM is an exposed interface, it should be comprehensible to customizers. At the moment, QGM is not easy to navigate, as it is implemented by a maze of C structures. However, conceptually it is a main-memory database. To make it truly useful to extenders, we should probably provide a high-level interface for querying this internal database.

Tailoring Semantic Optimization: When a new operation is added, the customizer may wish to specify how the new operation should interact with existing operations. He does this by specifying new rules for query rewrite. Customizers may also add new rules to improve the handling of existing operations, for example, to do duplicate join elimination [41] or to do magic sets optimization [54]. To add a new rule, customizers write two C functions (one for the IF condition and one for the action), and introduce them to the system.

Our rule-based approach to rewrite optimization assures both great flexibility and extensibility for new functionality, and is therefore very promising. Using C as our rule language gives the customizer great power. At the same time, managing the rewrite rules is complicated by several factors [43] which will be discussed in Section III-B2 below. Despite these complications, we find query rewrite a vast improvement over earlier systems. We have been able to try out a number of different transformations, so it certainly meets our goal of being able to experiment with new technology. We would strongly recommend a rule-based approach to others who are interested in semantic optimization.

Adding New Execution Strategies: A new execution strategy that can be composed from existing LOLEPOPs can be added to the plan generator's "repertoire" simply by defining that sequence of LOLEPOPs as a new alternative to the appropriate *STrategy Alternative Rule* (or STAR) in the plan generator. No recompilation or relinking of any Starburst component is required.[3] If a new strategy requires a new LOLEPOP, more effort is required. The customizer must define a property function for it, specifying any effect that that LOLEPOP has on the properties of its inputs, including the cost. In addition, the LOLEPOP requires a refinement function to convert its arguments' QGM references to external (Core descriptor) references, and of course an execution function that will be invoked by QES at execution time (see below). New properties may also be defined, either as a new component of the cost equation (e.g., the maximum number of buffers needed), or because a particular LOLEPOP affects or requires it (e.g., the merge join LOLEPOP requires a certain tuple order, and the SORT LOLEPOP achieves it). When a new property is added, only the property functions of LOLEPOPs that affect *that property* need be altered and recompiled; all others simply pass it through unchanged.

We are very pleased with the extensibility of the rule "language." Using this rule structure, we can readily express all the strategies of the R* optimizer, plus new strategies for composite inners [e.g., $(A*B)*(C*D)$], new join methods, new access methods, index ANDing, filtration methods such as semi-joins and Bloom-joins [37], dynamic creation of indexes on intermediate results, and materialization of tables at any point to force projection, all in under 20 rules. We believe that any desired extension, whether a new access method, join method, or new operation, will be easily expressible using STARs. We would like to have a parser for the rules that would allow them to be specified in a high-level language.

Executing New Operations: To add a new operator to the QES, the extender must define its data structure, write routines to interpret the operator, build the data structure at compile time, and display the data structure (for debugging). A customizer can usually exploit existing routines to do much of this. For example, if the operator's argument is a set of columns, an existing refinement routine converts that set to a list of field numbers in the Core descriptor; similarly, there is a routine to transform a set of predicates into a stack machine program for the expression evaluator. The hardest routine to produce is the one that interprets the operator. This is made easier by the common stream interface that each operator supports, allowing these functions to be developed independent of any other operators in the system. In fact, the addition of each operator that Starburst supports has been done as an extension to the existing system, and the simplicity of adding new operators has been verified.

Tuning Privilege Hierarchies: When a new operator is added, the customizer may wish to control the use of the operator by introducing a new privilege for the database objects on which the operator works. The Starburst authorization system [21] flexibly and efficiently supports granting and revocation of access privileges on database objects to users or groups of users (*authorization groups*). The authorization mechanism supports hierarchies of object access privileges, one hierarchy for each type of object, thus allowing the privilege hierarchy to reflect the requirements and semantics of the different object types. For example, the privilege hierarchy for tables is shown in Fig. 3. Users with the WRITE privilege automatically acquire the READ, INSERT, UPDATE, and DELETE privileges. Privilege hierarchies are themselves represented by (hierarchies of) "special" authorization groups. Adding a new privilege thus corresponds (roughly) to adding a new group. By storing the transitive closure of group memberships, the system can check user authorization in constant time, regardless of the complexity of the user or privilege group nestings.

We are quite pleased with the power and extensibility of this component. The authorization mechanism can be

[3]Currently, STARs are initialized using C assignment statements, so compilation and relinking *is* required. We plan to store STARs in catalog tables that can be updated by SQL queries and read as data when the system is initialized. Dynamic linking will still be needed to link user-defined functions referenced in STARs, notably the condition functions for each alternative.

Fig. 3. Privilege hierarchies for tables.

used to model an interesting variety of discretionary access control facilities, including recursively revokable grants and transfer of object ownership. The process of extending the authorization mechanism to control a new privilege is simple and straightforward. While storing the transitive closure of group membership may be quite costly in terms of storage space, this is not a scarce resource, and the size of the closure does not affect the cost of authorization checking.

2) Customizing Data Management:

Adding New Storage Methods and Attachments: New storage methods are added by first implementing a well-defined set of run-time operations such as scan, fetch, update, delete, insert, destroy table, and a compile-time property function to estimate access costs (for query planning). Each class of storage method operation has a vector associated with it, and a storage method implementor must append a new entry to each of these vectors. For example, the name of the function that performs relation scans for a new storage method is appended to the vector of scan operations. Attachments are added in a similar manner.

In many other DBMS's, such as System R, accesses and updates to access methods are hard-coded with the accesses to the associated table. For example, the call to insert a value into the index after inserting a record into a table is hard-coded, and adding a new access method requires locating all places in the code where appropriate calls need to be placed. The Starburst approach, with the use of the core relation descriptor and operation vectors, permits the addition of storage methods and attachments without modifying the code of existing storage methods and attachments.

The Core common services are extensible as well. For example, the logging and recovery service can be extended for a storage method or attachment by defining new recovery routines for DO (forward processing), UNDO (backward processing), and REDO (compensation processing). The lock manager can be used with its existing lock modes by simply defining new lock types, or it can be extended by defining new lock modes, lock types, lock compatibility matrices, lock request enqueuing and dequeuing, lock conversion and escalation, and lock deadlock detection and resolution. Alternatively, the customizer may implement his own service, but then he must ensure that it is compatible with the existing service.

We have used the data management extension architecture to add five different storage methods and five different attachments to Starburst. The types of relations implemented by the storage methods include three types of permanent relations, one that stores records in a "heap," one that stores them in a *B*-tree, and one that uses hashing, and two types of sequential temporary relations. The five attachments that have been implemented include one permanent and one temporary *B*-tree index manager, an attachment to enforce the "NOT NULL" column attribute, a signature attachment [9] used to locate records based on signatures of column values, and a precomputed join attachment that also manages multitable clustering.

With this experience, and ideas that we have for additional storage methods and attachments, we have recognized some of the benefits and weaknesses of the architecture. Our original implementation invoked all attachments only *after* calling the storage method to insert, delete, or update a record. We have since discovered applications that require more flexible timing of attachment invocations, e.g., *before* a record is actually inserted or modified, or before or after read operations. To date, we have only added the first of these, to support the precomputed join attachment that is described in Section IV. The addition of these automatic calls in the data manager was very simple, and the addition of similar calls to the scan and fetch operations can be done just as simply.

The order in which attachments are automatically invoked is currently determined by a systemwide ordering of attachment types that is based solely on the order in which the attachments were implemented. We have considered more complex ordering mechanisms in which attachment types and even attachment instances could be invoked in arbitrary orders that could be defined differently for each relation. At this point, we cannot justify the added complexity and performance degradation that such a mechanism would cause.

Reflecting Core Extensions in the DDL: Starburst's data definition language (DDL) processor has to process DDL statements for new kinds of objects with new kinds of attributes that are specific to that object, unlike the fixed DDL of previous systems. The challenge was to separate the generic parts that are required regardless of table or attachment type from the parts that are specific to individual storage methods or attachments, in both the DDL syntax and its processing. This was done in the syntax by allowing extension-specific attributes after the generic parts of the statement. For example, CREATE TABLE has the following format:

```
CREATE [storage_method] TABLE [tablegroup.]
  tablename
    (column_name_1 data_type_1 [,column_name_2
      data_type_2] · · · )
    [attributive_value_list].
```

An example of its use might be

```
CREATE BTREE TABLE Laura.Chocolate
    (Brand CHAR(15), Flavor CHAR(40), Rating
      INTEGER)
    KEYS (Flavor, Brand),
    CLUSTERED_WITH (Laura.Icecream).
```

This statement has only two keywords: CREATE and TABLE. The table name and a column list are required for any table, regardless of storage method, but all other

information, isolated in the attribute-value list, is specific to the storage method. The generic processing handles the required information in the statement, while the extension-specific processing interprets the attribute-value list. For example, the generic CREATE TABLE routine performs all the work that is required to create *any* table: it checks that the table was not defined previously, that the table's creator has the authority to name and create it, that there are no duplicate column names, that the columns' data types are legal, etc. It then passes control to the routines that are specific to the storage method, which check the attributes (KEYS and CLUSTERED_WITH in our example) and create Core and Corona descriptors for the table. Each attribute can be processed by its own routine, so that storage methods that share attributes can share their processing routines. Finally, the generic CREATE TABLE routine builds catalog records from the descriptors, and inserts them in the appropriate system catalogs.

We have extended the DDL to support several of the storage method and attachment extensions. On the whole, we have found it very easy to make the necessary changes. However, customizers may need more assistance with parsing the attribute-value list than we currently provide.

3) Further Support for Applications:

Defining New Datatypes and Objects: We expect one of the most common demands of application designers will be for the addition of new datatypes to the system. New types can be extremely useful to the user writing queries, as they allow type-checking to better reflect the user's semantic intent, and enable better structuring of the user's data. We plan to allow the definition of almost any type at the field level, and to allow columns whose type is externally defined to appear anywhere a column with built-in type can appear. An overview of our approach to externally-defined abstract datatypes appears in [52]. Our extensible type system is not yet operational. At the moment we support only a fixed set of base types, and incorporating a more extensible system will involve some substantial changes to current Starburst code.

We also plan to provide support for *complex objects*, that is, objects which span multiple records of one or more relations. We have defined a language that allows complex objects to be specified as *complex views* over the stored relations (or over normal or other complex views). We are currently working on efficient support for retrieving and updating these complex views.

There are a number of open questions in this area. We would like to extend the complex view mechanism to allow us to define not just individual complex objects, but classes of objects, and to somehow unify this type system with the field-level type system. We would like to better understand the relationship of this work to work on "object-oriented" database systems, and be able to compare them in terms of modeling power and ease of use. We are clearly in need of concrete experience with specific applications to give us at least a preliminary feel for these issues.

Efficiently Supporting Ordering Functions: We are making the sort component of Starburst more extensible by providing several exits to customizer-defined functions. Objects of different types may require different comparison functions; e.g., lexical compare is not much good for comparing dates. When a comparison function is provided as part of a type definition, it will be added to a vector of comparison routines, to be automatically invoked during sort. Customizer-defined functions will be invoked when duplicate values are detected to accomplish a range of functions and enhance performance. A routine to just throw away the duplicate tuple achieves the DISTINCT function; functions to accumulate aggregate totals (such as SUM or COUNT) perform aggregation. In either case, tuples are progressively removed from the sort, reducing the length of runs and possibly the number of runs. This in turn can reduce the number of merge passes needed [5]. More sophisticated routines could be used to support the efficient evaluation of more complex functions, such as the selection of the N highest values, or the Jth percentile. A customizer-defined routine would exploit the knowledge of the number of tuples or their range to reject tuples whose values lie beyond the specified boundary as early as possible.

Preliminary results indicate that sort will be efficient as well as extensible. We believe that the design is flexible enough to be tuned to the particular applications it supports, for even greater efficiency.

Expressing More Application Semantics: In Starburst, we want to add a facility that allows users and application programs to define production rules: rules specifying that certain data manipulation operations are to be automatically executed whenever certain conditions are met. Production rules may be useful for integrity constraint enforcement, view materialization, authorization checking, active databases, and knowledge-base and expert systems. A simplified form of "triggers" and consistency constraints was proposed for System R [25].

We have developed a new syntax and semantics for production rules in relational database systems. In keeping with the set-oriented approach of relational data manipulation languages, the proposed production rules are also set-oriented: they are triggered by sets of changes to the database and can perform sets of changes. The condition and action parts of the production rules can refer to the current state of the database as well as to the sets of changes triggering the rules. A syntax for rule definition is defined as an extension to SQL. A model of system behavior is used to give an exact semantics for production rule execution, taking into account externally-generated operations, self-triggering rules, and simultaneous triggering of multiple rules. As a first step towards implementation, a rule execution algorithm has been designed that reflects the desired semantics [50]. It is too early in the implementation to draw conclusions about our design of the rule system, but we are encouraged by the power it provides and anxious about the bookkeeping needed to maintain the information used for rule triggering and execution.

B. Mechanisms Used for Extensibility

For simplicity and ease of extension, a uniform mechanism for extending the system at all levels would have been ideal. Unfortunately, this does not exist in Starburst—nor is it clearly achievable. However, in looking over our design, we find two general mechanisms were frequently used. Vector lists are used throughout Starburst to guarantee unlimited system growth; rule languages were also popular as they provide great flexibility for altering the system's logic. We comment on our use of these two mechanisms below.

1) Vector Lists: Vectors of functions and values have been used extensively in Starburst as a mechanism to permit easy extensibility. For instance, all of the data manager operations are supported by vectors of functions for their various storage method and attachment implementation. An example illustrates the popularity—and usefulness—of this technique. When an insert to a relation is performed, the Core relation descriptor is examined to determine the storage method and attachments that must be called. First, a vector of preinsert attachment routines is searched to see if any attachments defined on the relation should be called before the record is inserted into the relation. Then the vector of storage method insert routines is used to route the record to the correct storage method to perform the actual relation insert. After the insert, the data manager scans the relation descriptor looking for attachments that are defined on the relation. For each such attachment, the post-insert routine for the attachment is located in yet another vector and is called with the inserted record value and TID information that was returned by the storage method.

Vectors are used to facilitate the addition of new implementations of interfaces. They have also proved useful for extending existing interfaces. For example, when we realized that we needed premodification attachment calls to support a clustering mechanism, a new vector of preinsert functions was defined. None of the existing attachments needed a premodification function, so the entries for those attachments were simply set to NULL. This allowed the interface to be extended, without requiring changes to any of the existing attachments. Vectors of Booleans were used to communicate storage method and attachment properties when we discovered that there could be dependencies between these (see Section IV-C).

2) Rules, Rules, Rules!: Rules provide a high-level and flexible interface for modifing the DBMS's logic. Starburst uses rules internally in the optimizer both for query rewrite and for plan optimization, and also processes production rules written by users to specify constraints to enforce, or actions to trigger, as a result of changes to the database. Since these are quite distinct functions, we have developed rules and rule processors that are specialized to each function. The query rewrite rules, written in C and compiled with Starburst, are context-dependent. By contrast, the context-free plan generation rules are input as data to Corona and interpreted during query compilation. The production rules are an extension to Hydrogen that are processed by Corona [50].

Using a general-purpose rule language such as Prolog for the optimization rules was considered and rejected. Besides the obvious problems of availability of compatible compilers on the RT, how could we access and modify complex C data structures such as QGM and QEP's from such a language? While the unification capabilities of Prolog would have simplified our code, its poor performance in a prototype query optimizer by Graefe and DeWitt [23] reinforced our belief that a specialized rule processor could better exploit the structure inherent in this application. To experiment easily with different search strategies, we needed more control of the order of rule execution than allowed by Prolog's depth-first evaluation of alternatives and its CUT operator [30].

Implementing a separate rule processor for each type of rule has added considerable complexity to Starburst. Some problems have had to be solved several times in different ways. For example, the order in which rules are evaluated during plan generation is controlled by a prioritized queue. In query rewrite, the order in which rules are executed is controlled to a lesser degree by defining rule classes and limiting unification to a class of rules, so that one type of transformation (e.g., view merging) can be inhibited until another (e.g., predicate pushdown) completes. Other problems with rules must still be addressed. For example, for each rule type, we would like to have mechanisms to test the legality and safety of new rules (when possible), and ways to detect and/or prevent looping. More challenging still are tools to help the rule designer minimize interactions and redundancy between rules, and to maximize rule reusability, seeking to avoid an exponential growth in the number of rules when extensions are added (see [43] for a discussion of some of these issues in the context of query rewrite).

Virtually all of these problems are not peculiar to our application, but are in fact generic to all rule systems. Developing several different rule processors has allowed us to experiment with various approaches to solving some of them, but obviously we have much left to learn.

We would like, for simplicity, to be able to reduce the number of rule systems in Starburst. The presence of two rule systems within the optimizer alone is particularly unsatisfactory. There are two types of rules for a customizer to understand, two rule engines to maintain and enhance. Getting the two to work together efficiently raises several interface issues that we have not resolved yet. Will the "CHOOSE" alternatives remaining after semantic optimization proliferate unmanageably as more rules are added? Will we need to "ping-pong" between semantic and plan optimization? How well do both optimizations really perform? Ideally, we would like to find some unified set of rules that could be input as data to the optimizer, yet could be interpreted efficiently by a single rule processor. Could this more general form of rules even subsume our production rule language, allowing us to support knowledge-base applications and a more "active" database [19]?

IV. Experience with Extensions

Several extensions have been integrated into Starburst. The choice of extensions reflects the personal research interests of the customerizers rather than any particular notion of global importance. In this section, we report first on two extensions by our team members: the signature attachment, which demonstrated the extensibility of the data manager, and the addition of an outer join operator, which exercised the extensibility of the language processor. Our first external database customizers extended Starburst this past summer; we chronicle their experiences in some detail since we discovered, through their work, several blind spots in our design. Finally, we take a look at how well Starburst has achieved its goal of extensibility.

A. Adding a Signature Attachment

A signature index [9] is a special type of index that provides an efficient access path for conjunctive multifield queries and that could support predicates over abstract data types such as text or chemical spectra. The signature index was implemented by incorporating multilevel signature coding techniques within the framework of our existing *B*-tree access method. In our approach, one or more signature fields are appended to the leaf and parent entries within the index. Signature entries in interior pages serve to filter entire groups of lower level signatures, while leaf signatures filter specific data objects during predicate evaluation.

The signature index attachment was added by first specifying the attachment's appearance and behavior at the Hydrogen, Core, and Corona levels. An example of the CREATE statement (Fig. 4) shows the attributes needed to create a signature index. The COLS clause specifies which fields of the relation are to be used as normal *B*-tree index columns. The SIGCOLS clause specifies which columns are used in the signature. After determining what attributes were needed, we designed the Core and Corona descriptors. Then we implemented an "interpreter" for the new attribute-value pairs to be invoked during "CREATE signature_index." The interpreter parses the pairs, tests semantics, and generates Core and Corona descriptors for the signature attachment type. Using a standardized interface, we defined the basic operations possible with this attachment, wrote the routines for these operations, and "plugged them in." In the plan optimizer, a new value for the "access method type" argument to the SCAN LOLEPOP was added. No rules were changed. A function to determine which predicates can be applied by this attachment was adapted from that for *B*-tree indexes, to which it is very similar. Eventually, a function for estimating costs will be needed.

The whole process was gratifyingly simple and straightforward. Most of the new code implemented the actual attachment operations (scan, update, delete, etc.). The DDL extension was very easy (accomplished in half a day), and the changes for plan optimization were trivial—though that did not include the function to estimate the cost of operations on the attachment. In other words, while designing and implementing the signature indexes took months, hooking them up as an attachment to Starburst took about a day. Other data management extensions have been accomplished equally easily.

```
CREATE SIGNATURE_INDEX BrandFlv
    ON Laura.Chocolates
       COLS (Brand, Flavor),
       SIGCOLS (Brand, Flavor)
```

Fig. 4. Example of create signature index.

B. Adding an Outer Join Operation

As a test of the language processor's extensibility, we have designed and partially implemented an extension to support a left outer join operation. At the language level, we defined a new table function whose result is the left outer join of the input tables. Although syntactically a new operation looks like a function call, table functions require sophisticated processing internally. Many of Corona's built-in functions (e.g., catalog lookup, name resolution, most of the semantic checking, etc.) can be used to generate QGM for queries containing left outer joins. The customizer must write code for those parts that are different. In this case, we wrote code to check whether the left outer join was associative or commutative (in general it is neither). Also, the meaning of any predicates will be different, because if matching inner values are not found, the outer tuple still must be included in the result. This difference had to be flagged in the QGM. Many of the query rewrite rules for normal joins can be used for outer joins, with some important differences. For example, we had to write one new rule and modify one other to specify when predicates should be applied.

We did not implement the rest of this extension, but we did complete the design. For the plan optimizer, the major difference between left outer joins and regular joins is that outer joins are not (normally) commutative. This could be accounted for by adding an IF condition to the rule which constructs the join permutations, to ensure that the only order explored is the valid order. However, it would probably be more efficient to write a simplified join enumerator for outer joins that exploits the order given by the query. In addition, the rules will need to invoke a different operator for each method for implementing the left outer join, much as we now have distinct operators for the nested-loop, sort-merge, and Grace hash join methods [35]. Finally, for QES, left outer join is just a new operator, requiring a new generation routine, a routine to interpret the operator (perform the left outer join), and a routine to print the operator for debugging.

The hardest part of the outer join extension was keeping track of all the different places that changes had to be made, and all the different types of extensibility involved. In fact, the design of the extension required two team members who were expert in different parts of the system.

Clearly, better support will be needed before an external customerizer will be able to extend the language processor.

C. An External Customerizer's Experience

Up to this point, we have regarded extensibility from the perspective of the team building the Starburst prototype. Eventually, extensions must be built and easily integrated by a different set of people, the customizers. This is an important measure of the success of the extensibility concept.

During the summer of 1989, our first external customizers (visitors from the EXODUS project [7] at the University of Wisconsin) developed an attachment to do precomputed join maintenance and multitable clustering for parent/child table pairs. The goal was to provide a fast access path via record "pointers" for frequently needed joins. For example, assume we have a pair of tables DEPT (dno, dname, · · ·) and EMP (eno, ename, dno, · · ·), where the relationship between DEPT and EMP instances is one-to-many. This attachment maintains hidden "pointer" fields to allow fast DEPT.dno = EMP.dno joins. It uses these fields to wire the "record families" together in a Codasyl-like manner. The attachment requires a unique index on Dept.dno; it uses this as its access path for quickly finding the parent tuple for newly inserted child tuples. The precomputed join attachment provides various degrees of clustering, either FULL (to cluster related DEPT/EMP tuples together), CHILD (to cluster the EMP tuples for a given DEPT together), or NONE (for no clustering).

This was a complex attachment, well beyond what the original designers of the data management extension architecture had anticipated. As a result, several architectural issues arose, and some of the early decisions needed to be revisited. The implications for extensibility are related here.

One of the first things discovered by the visiting group was that the data management extension architecture contained several hidden assumptions. There was, first of all, an assumption in the catalog design that attachments were attached to one single (or distinguished) table. Obviously, this was not true for the precomputed join attachment, which of necessity spanned two tables. This issue was sidestepped in the implementation by "officially" attaching an instance of a precomputed join attachment to the parent table (in the catalogs—in Core it was attached to both tables, of course!). However, this approach is not entirely satisfactory; a query to the attachment catalog will currently not see the child side of each attachment, so in our example the EMP table would appear to have no attachments. In addition, the meta-data for the tables had to be modified to introduce the notion of "dependent descriptors," so that an attachment would have a chance to ensure that all of the schema information about all of the tables involved in a given update would be present in the query plan for update queries.

Precomputed join is not the only multitable attachment we can imagine. Other examples include an attachment that does incremental view maintenance, an attachment that enforces an N-table integrity constraint, and an attachment that implements an N-table production rule. Thus, it might have been better to have changed Starburst so that these types of attachments would clearly appear to attach to more than one table, rather than sidestepping that issue. Time constraints prevented a more satisfactory solution to this problem.

Starburst also assumed that each attachment type is independent. However, there are at least two ways in which an attachment may be dependent on another attachment or on a storage method. An attachment may depend on certain properties of the table's storage method (e.g., clustering), or on certain properties of one or more other attachments. These properties may be of the attachment type, or even of a particular attachment instance. For example, to do CHILD clustering for the precomputed join attachment the child storage method must support clustering based on Core's clustering hint facility, and for FULL clustering, the parent and child storage method instances must be set up to allow records from the two tables to be clustered together. An attachment may also depend on the (continued) existence of another attachment. For the precomputed join, for correctness and efficiency, the parent table must have an attachment that allows fast lookups on the parent key and ensures that the parent key is in fact unique. To deal with these problems, the Wisconsin customizers added vectors of Boolean values and Boolean functions that allow storage method and attachment properties to be communicated in a clean manner.

As we mentioned above, the data management extension architecture originally invoked attachments only after modifications (inserts, updates, or deletes) to the relation, not before. However, attachments may actually need to have access to a record before it is modified as well. One case where this makes sense is for clustering: an attachment that wants to suggest a disk location for the record cannot help much if the record has already been assigned a home on disk. Another case is an attachment that needs to set or modify certain fields of the record, which is obviously better done before the record goes out to disk. The precomputed join attachment was an example of both cases. Other examples can also be thought of: for example, any clustering index-like attachment such as a clustered B-tree index, or attachments that do some sort of compression or encryption of the data in a table. To handle these types of attachments, premodification calls were added to the architecture.

Finally, Starburst provided no assistance to attachments that may need to alter the internal schema for one or more tables. For example, the precomputed join attachment added several hidden record "pointer" (TID) fields to the parent and child tables. Another example might be an attachment that transparently maintained some sort of virtual column. Minor reorganization of the Starburst catalog management code was required to enable these sorts of alterations.

In addition to these architectural issues, our customizers found a few flaws in Starburst's software engineering that hindered efficient development of their extension. The most annoying involved our "uniform" record structure. Records must be aligned and there are several different ways of padding them to achieve this. Different components of Starburst took different approaches. This required the poor customizers to understand each component's interpretation of the record structure and use the appropriate versions of the record manipulation routines. Fortunately, this can easily be fixed by "scrubbing" the code to make all components consistent.

We have learned a lot from our first customization experience. On the positive side, the extension, although more complex than any extensions we had anticipated, was accomplished. However, alterations to the extension architecture and the basic Starburst code were needed. These alterations have undoubtedly left Starburst in better shape for handling future extensions. However, further additions to the architecture are probably necessary before Starburst is ready for the most general conceivable (multitable) attachments. For example, it is likely that we will find attachments that must be invoked either before or after a record is read (e.g., for various auditing or security functions). While we do not anticipate that these calls will be hard to add, there may well be more difficult ones we have not yet discovered. All in all, our experience, while a constructive one which has increased Starburst's extensibility, has also been a humbling one, making us realize just how ambitious and elusive our goal is.

D. Is Starburst Sufficiently Extensible?

Starburst is currently extremely extensible. We know of no other system that can be extended as easily, in as many different ways. Extensions have been done both by members of the Starburst team and by external customizers. Extensibility is pervasive; all components are extensible, though some are clearly more extensible (extensible in more ways) than others. Extentions that are clearly confined to a single component (or even a couple of components) are reasonably quick and painless.

However, Starburst is not perfect, as we cannot anticipate all possible types of extensions. Thus, our first customizers chose a class of extensions for which parts of Starburst were unprepared, forcing us to modify our design and code. This was easily accomplished at this stage of our prototype, but such modifications may not always be feasible in the future, and this could bound Starburst's extensibility.

We could also make it much easier to develop extensions to Starburst. Extensions that span several components cannot currently be successfully developed and integrated without a thorough knowledge of the system's architecture. Adequate tools are needed before customizers can safely and independently add these types of extensions to the system. In particular, we need tools to help customizers determine *where* they must extend the system and *how* to integrate the extension. *Where* is difficult because most extensions require changes to several components. However, many extensions will tend to fall into classes, e.g., the addition of a new access method, the addition of a new operator, etc. Each class will typically require certain types of extensions to a fixed set of components. Thus, it should be possible to create tools (or at least documentation) to guide most customizers.[4]

How is difficult in large part due to the inconsistent level of documentation and to the state of the Starburst code. The code has been written by many past and present members of Starburst who have different programming styles, variable naming conventions, different conventions for include and source files, different attitudes towards commenting code, and different documentation practices. Some of these practices are better than others, and the variations alone increase the difficulty in understanding the code. Starburst's extensibility could be improved by better documentation and more standardization of coding practices, but it is not clear how much effort in this direction should be made in a research environment where the balance between independence and standardization is much different than one would choose in a product development environment. More tools to aid the customizer and distance him from the Starburst code base might circumvent this problem.

In summary, we believe that Starburst is sufficiently extensible to support most applications. There will always be some that will require changes to the base system, but that number should be quite small. More work on tools, polishing the code, and documentation would make Starburst even easier to extend.

V. The Building of Starburst

A. What We Did . . . and How

The Starburst system currently consists of approximately 100 000 lines of C source code in over 400 source files. Slightly less than one half of the source lines are for Core functions, the remainder implements Corona functions. Core is fully functional, though many components have yet to be tested rigorously. Concurrency control has only recently been integrated and has received extremely limited testing. Corona can now compile and execute single-table queries, and nested block joins. Support for more complex statements (especially other join methods and recursion) is being added to the plan optimizer and query refinement. Other parts of Corona are more complete, but all are missing some function. The most glaring is the absence of the abstract type facility, which is partially designed, but not at all implemented. We have a simple application program interface, and a substantial driver environment.

Starburst currently runs on the IBM RT/PC, a RISC-based workstation, under the AIX operating system (based

[4]In fact, our Wisconsin customizers have written a guide for the addition of new attachments.

on Unit V). It is written in the C programming language. Early in Starburst, we made the conscious decision to switch from our IBM mainframe, PL/I-like programming environment to the brave new world of workstations. This choice made it difficult to leverage our existing DBMS codebases. While some of the algorithms for storage and access methods were borrowed and extended, all the Starburst code was written from scratch. As we designed and built new components for Starburst, we attempted to implement pieces of the system so that they would be independently reusable. For example, the index component can operate as a stand-alone AIX utility.

Over the last few years, the RT/PC environment has grown increasingly comfortable and reliable. But during the early phases of Starburst, two researchers (approximately 20% of our team) devoted nearly full time to maintaining and debugging the programming environment and tools. A C language macro trace preprocessor was written to circumvent the limitations of the existing preprocessor, a run-time trace and structure display facility was built to provide trace control and debugging at a higher level than symbolic debuggers, and an error architecture was designed for reporting error events at all levels of the system. In addition, a large number of scripts were written to handle linking, compilation, and compilation errors for various editor environments, and customized programs were developed to manage file transfer functions such as check-in and check-out for source and object files between user machines and our various workstation and mainframe servers.

B. History and Administration

Starburst began in 1985, during the final stages of R*. As many of the original team members were still involved in R*, we have had, from the beginning, a challenging task: to advance the state of Starburst while accommodating the diverse research interests of key team members. To further complicate matters, the broadness of our goal of extensibility made it hard to find a crisp focus for our prototype.

From the outset, Starburst was organized into a Core group and a Corona group, under two managers. Team membership has varied over time, but has generally consisted of about ten researchers, with a small amount of programming support in the form of summer students. The two groups were soon at very different stages of prototype development. Most work on Core was completed over a year ago, while Corona has only recently been able to handle any DML statements. This was in part due to research styles. In addition, there was significantly more pressure to produce a working Core, as Corona relied on it, and it was somewhat easier, as extensibility in Core (the data management extension architecture) was more clearly defined and bounded earlier on. Finally, several members of the initial Corona team were heavily involved in R* until almost a year after the start of Starburst. The different rates of progress have been occasionally frustrating for many team members, especially (but not exclusively) for those in Core. To compensate, Core people lent their skills to Corona, or became involved in applications or technologies that could motivate extensions to Starburst. This led to a diffuse atmosphere leading to a certain loss of momentum for Starburst.

On the other hand, the diversity of research interests present in the Starburst team has been a valuable resource for Starburst. Starburst is more extensible and more efficient because team members pursued their interests in various applications (e.g., logic programming, molecular chemistry, expert systems) and technologies (e.g., access methods, buffer management). Furthermore, these "outside" interests may have kept the team together through times in which members might easily have drifted away to other projects, with their skills being "lost" to Starburst.

VI. Performance

Currently, the results of compiled SQL queries can only be displayed on the terminal and cannot be inserted into temporary relations. Thus, we executed a subset of the Wisconsin Benchmarks through a QES test program that takes as input a description of a query execution plan, and can insert query output into a temporary relation. Fig. 5 shows the results of seven of the benchmarks. We have included, as a reference point, the numbers for University INGRES on the VAX 11/750 running Berkeley Unix 4.1 [48], although it is a rather unfair comparison given the radically different architectures, disk speeds, etc. The benchmarks include scans of 10 000 record relations without using an index, with a clustered index, and with an unclustered index. One set of benchmarks inserts 1% of the records of the relation into a temporary relation, and another set inserts 10%. The seventh benchmark inserts all of a 1000 record relation into a temporary relation. These are very early results, and were run with our debugging code enabled and no tuning on an IBM PC-RT under the AIX operating system. The buffer pool size was 1 megabyte and it was flushed before each run so no useful pages were in the buffer pool at the start of any of the benchmarks.

We feel that these are reasonable numbers given the processor, disk, and operating system. For example, we use the AIX file system which does not guarantee contiguous allocation of disk blocks and transfers data in 2K blocks resulting in two I/O's per Starburst page. Our record structure results in fewer records per page than other schemes. We have not implemented prefetching, which might compensate for these inefficiencies. These are reasonable design decisions for a research prototype.

VII. Future Directions

At mid-flight, we have identified several research directions to focus on. In the short term, emphasis will be put on the critical pieces that are missing to make our system fully operational and attractive to a wide range of applications. Next, we need to study all the novel implications of extensibility on a running system. In the long

	BENCHMARK	TIME (sec) Starburst	Univ. INGRES
1	(1%, no index)	23	53.2
2	(10%, no index)	37	64.4
3	(1%, clustered index)	2.5	7.7
4	(10%, clustered index)	23	27.8
5	(1%, unclustered index)	7.3	59.2
6	(10%, unclustered index)	65	78.9
19	(all of 1000 record relation)	20	--

Fig. 5. Wisconsin Benchmark results.

term, we plan to use Starburst as *the* platform for our future database research.

Considerable effort is now required in the language area. Is embedding Hydrogen in a standard programming language still the most flexible approach, or should Hydrogen be extended to define the usual procedural constructs with the notion of "persistence"? How best do we reconcile the incompatibility of set-oriented Hydrogen and instance-oriented programming languages? "Objects" are manipulated by programming languages, Hydrogen, and Starburst in a nonhomogeneous way: this leads to costly interfaces, several representations of the same object, and decreased performance. What programming languages are required to manipulate "complex objects"? How can we optimize accesses to those objects, and quickly materialize them in response to queries? How close should the relationship be between the data model and the application programming interface? We are actively seeking a closer collaboration with experts in programming languages for help in this area.

Once our system is in "production mode," and some number of extensions exist, the system will continue to evolve due to new technology and the addition, modification, and removal of extensions. The management of these extensions is in itself a worthwhile research topic. How can we make the evolution of extensions in the system as transparent as possible to applications? Clearly, customizers can make mistakes, and some extensions could potentially damage the kernel system. How can we guarantee a safe kernel system? Even if the kernel is safe, how can we help to determine who is to blame (extension or kernel) when something goes wrong? An extensible system should be able to handle several extensions at a time. How can we detect when extensions may conflict? How will conflicting extensions be resolved? We will need improved tools to help customizers ensure that their extension is complete and viable (e.g., was the authorization component extended as part of this extension?), and to improve their awareness of existing extensions, so that they can avoid conflicts and even reuse parts of appropriate existing extensions. We will need some way of predicting the effect that a new extension will have on system performance, and of monitoring performance to contain the damage done by "bad" extensions (e.g., the introduction of a bad rule). We will need a way of dynamically linking extensions into a running system. And eventually, we must examine the question of extensibility in a distributed heterogeneous environment: how will meta-data about extensions be exchanged between sites? Is this a simple extension for query planning? These are some of the open questions to be answered.

Our long-term goal is to use Starburst as our principal platform for research in database systems. Here, we present a partial list of topics that we plan to investigate, using Starburst as a testbed. Main memory databases: since the main memory of workstations is rapidly expanding, keeping entire relations in main memory becomes a possible alternative; most algorithms need be reconsidered to take advantage of this feature. Parallelism: so far Starburst does not take advantage of multiple processes being able to concurrently participate in the processing of a query: this ambitious extension will definitely test a great deal of our global architecture. Data types and complex objects: is a seamless handling of objects by the programming languages and the DBMS achievable? User-defined rules: now that we have defined a language for user-defined rules we must extend Starburst to support it. We should revisit our three separate rule systems and see if we can find one which would be sufficient for all three needs.

VIII. Conclusions

From its conception, Starburst was envisioned as a long-term project intended to become a prime vehicle for exploratory database research well into the 1990's. In the first phase of the project, we focused on two major themes: to build a cleaner and more modular relational DBMS kernel, and to provide an extensible DBMS framework that could easily integrate a variety of new requirements. Richer internal data structures, greater integration and interchangeability of components, and some engineering for performance greatly contributed towards the success of the first objective. We learned several lessons while pursuing the extensibility goal. First, the power of the rule systems used in optimization considerably enhanced Starburst's extensibility. Second, Starburst's extensibility is not without certain limitations: e.g., even though our data management extension architecture was very general, certain aspects of our implementation made it tricky to have an attachment spanning multiple tables. Third, although every component is extensible, very different approaches to extensibility were taken, making Starburst more difficult to understand and extend.

There is still a long way to go before the Starburst system can really be used and extended by independent applications and customizers. We believe that extensible systems will be extremely important both for research and in the commercial realm. We are already getting requests from potential applications of all sorts, both from other researchers and from customers. We believe, although we cannot yet prove, that by making a relational system highly extensible, we will vastly expand the set of applications a relational system can satisfy. On the other hand, other technologies also hold a great deal of promise. We are not sure that even the most extensible relational database management system will be as adaptable as an object-oriented DBMS. This is one of the questions we plan to explore. There is much exciting research ahead of us, and we feel very fortunate to have Starburst as a tool for exploring these diverse and challenging topics.

Acknowledgment

A system such as Starburst requires the efforts of many people. We gratefully acknowledge the contributions of all members of the Starburst team, past and present, who have made Starburst what it is today. Besides the authors, their ranks include B. Cody, S. Finkelstein, C. Freytag, R. Gagliardi, W. Hasan, M. Lee, T. Lehman, C. Mohan, I. Mumick, H. Nguyen, K. Ono, K. Rothermel, U. Schreier, P. Schwarz, J. Widom, and, last but certainly not least, B. Yost. Several people have helped us greatly in the writing of this paper. J. Widom provided the paragraphs on production rules. I. Traiger and U. Schreier constructively read a draft of this paper. We are also very grateful for the many constructive comments of the referees. This paper was produced with the able assistance of D. Sandman.

References

[1] S. Abiteboul, M. Scholl, G. Gardarin, and E. Simon, "Towards DBMSs for supporting new applications," in *Proc. 12th Int. Conf. VLDB*, Kyoto, Aug. 1986, pp. 423-435.

[2] F. Bancilhon *et al.*, "The design and implementation of O2, An object-oriented database system," in *OODBS2 Workshop Proc.*, Badmunster, RFA, 1988.

[3] J. Banerjee, W. Kim, H. J. Kim, and H. Korth, "Semantics and implementation and schema evolution in object-oriented databases," in *Proc. ACM SIGMOD 1987*, San Francisco, CA, May 1987.

[4] D. Batory, "GENESIS: A project to develop an extensible database management system," in *Proc. 1986 Int. Workshop Object-Oriented Database Syst.*, Asilomar, CA, Sept. 1986.

[5] D. Bitton and D. DeWitt, "Duplicate record elimination in large data files," *ACM Trans. Database Syst.*, vol. 8, no. 2, pp. 255-265, June 1983.

[6] D. Bitton, D. DeWitt, and C. Turbyfill, "Benchmarking database systems, A systematic approach," in *Proc. 9th Int. Conf. VLDB*, Nov. 1983, pp. 8-19.

[7] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS extensible DBMS project: An overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, Eds. Los Altos, CA: Morgan-Kaufman, 1989.

[8] H. K. Chang, "Compressed indexing method," *IBM Tech. Disclos. Bull. II*, no. 11, Apr. 1969.

[9] W. W. Chang and H. J. Schek, "A signature access method for the Starburst database system," in *Proc. 15th Int. Conf. VLDB*, Amsterdam, Aug. 1989.

[10] W. W. Chang and C. Mohan, "Front key compression with efficient binary search," paper in progress 1989.

[11] H. T. Chou and D. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proc. 11th Int. Conf. VLDB*, 1985, pp. 127-141.

[12] D. Comer, "The ubiquitous B-Tree," *ACM Comput. Surveys*, vol. 11, no. 2, June 1979.

[13] R. Crus, "Data recovery in IBM Database 2," *IBM Syst. J.*, vol. 23, no. 2, 1984.

[14] P. Dadam, K. Kuspert, F. Andersen, H. Blanken, R. Erbe, J. Gunauer, V. Lum, P. Pistor, and G. Walch, "A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies," in *Proc. ACM SIGMOD*, Washington, May 1986, pp. 356-367.

[15] D. Daniels, "Query compilation in a distributed database system," IBM Res. Rep. RJ3423 IBM Research Lab., San Jose, CA, Mar. 1982.

[16] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. Wilms, "An introduction to distributed query compilation in R*," in *Proc. Second Int. Conf. Distributed Databases*, Berlin, Sept. 1982. Also available as IBM Res. Rep. RJ3497, San Jose, CA, June 1982.

[17] C. Date, "A critique of the SQL database," *ACM SIGMOD Rec.*, 1984.

[18] U. Dayal and J. M. Smith, "PROBE: A knowledge-oriented database management system," in *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Brodie and Mylopoulos, Eds. New York: Springer-Verlag, 1986.

[19] U. Dayal, "Active database management systems," in *Proc. 3rd Int. Conf. Data Knowledge Bases*, Jerusalem, June 1988, pp. 150-169.

[20] K. P. Eswaran and D. D. Chamberlin, "Functional specification of a subsystem for database integrity," in *Proc. 1st Int. Conf. VLDB*, Framingham, MA, Sept. 1975.

[21] R. Gagliardi, G. Lapis, and B. Lindsay, "A flexible and efficient database authorization facility," IBM Res. Rep. RJ6826, San Jose, CA, May 1989.

[22] G. Graefe and K. Ward, "Dynamic query evaluation plans," in *Proc. ACM SIGMOD*, Portland, OR, May 1989, pp. 358-366.

[23] G. Graefe, "Rule-based query optimization in extensible database systems," Ph.D. dissertation, Univ. of Wisconsin, Madison, WI, Aug. 1987.

[24] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible query processing in Starburst," in *Proc. ACM SIGMOD*, Portland, OR, May 1989, pp. 377-388.

[25] W. Hasan and H. Pirahesh, "Query rewrite optimization in Starburst," IBM Res. Rep. RJ6367, San Jose, CA, Aug. 1988.

[26] *OS/VS Virtual Storage Access Method (VSAM) Planning Guide*, Order No. GC26-3799, IBM, Armonk, NY.

[27] *OS/VS Virtual Storage Access Method (VSAM) Logic*, Order No. SY26-3841, IBM, Armonk, NY.

[28] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surveys*, vol. 16, no. 2, pp. 111-152, June 1984.

[29] M. Kitsuregawa, H. Hanaka, and T. Moto-oka, "Application of hash to database machine and its architecture," *New Generation Comput.*, pp. 62-74, 1983.

[30] M. Lee, J. C. Freytag, and G. M. Lohman, "Implementing an interpreter for functional rules in a query optimizer," in *Proc. 14th Int. Conf. VLDB*, Long Beach, Aug. 1988, pp. 218-229. Also available as IBM Res. Rep. RJ6125, San Jose, CA, Mar. 1988.

[31] M. Lee, "Interaction between the query processor and buffer manager of a relational database system," Masters Thesis, MIT, May 1989. Also available as IBM Res. Rep. RJ6884, San Jose, CA, June 1989.

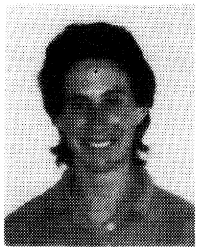
[32] B. Lindsay, J. McPherson, and H. Pirahesh, "A data management extension architecture," in *Proc. ACM SIGMOD '87*, San Francisco, CA, May 1987, pp. 220-226. Also available as IBM Res. Rep. RJ5436, San Jose, CA, Dec. 1986.

[33] G. M. Lohman, D. Daniels, L. M. Haas, R. Kistler, and P. G. Selinger, "Optimization of nested queries in a distributed relational database," in *Proc. 10th Int. Conf. VLDB*, Singapore, 1984, pp. 403-415. Also available as IBM Res. Rep. RJ4260, San Jose, CA, Apr. 1984.

[34] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. Wilms, "Query processing in R*," in *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory Eds. New York: Springer-Verlag, 1985. Also available as IBM Res. Rep. RJ4272, Apr. 1984.

[35] G. M. Lohman, "Grammar-like functional rules for representing query optimization alternatives," in *Proc. ACM SIGMOD*, Chicago, May 1988, pp. 18-27. Also available as IBM Res. Rep. RJ5992, San Jose, CA, Dec. 1987.

[36] R. A. Lorie and J. F. Nilsson, "An access specification language for a relational data base system," *IBM J. Res. Develop.*, vol. 23, no. 3, pp. 286–298, May 1983.
[37] L. Mackert and G. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *Proc. 12th Int. Conf. VLDB*, Kyoto, Aug. 1986.
[38] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," IBM Res. Rep. RJ6649, Jan. 1989, IBM Almaden Research Center, submitted to *ACM Trans. Database Syst.*.
[39] K. Ono and G. M. Lohman, "Extensible enumeration of feasible joins for relational query optimization," IBM Res Rep. RJ6625, San Jose, CA, Dec. 1988.
[40] S. Osborn, "Extensible databases and RAD," *IEEE Database Eng.*, vol. 10, no. 2, pp. 10–15, June 1987.
[41] N. Ott and K. Horlander, "Removing redundant join operations in queries involving views," IBM Int. Rep. TR 82.03.003, IBM Heidelberg Scientific Centre, Germany, Mar. 1982.
[42] H. Paul, H. Schek, M. Scholl, G. Weikum, and U. Deppisch, "Architecture and implementation of the Darmstadt database kernel system," in *Proc. ACM SIGMOD*, San Francisco, CA, May 1987, pp. 196–207.
[43] H. Pirahesh, "Early experience with rule-based query rewrite optimization," in *Proc. Workshop Database Query Optimization*, Portland, OR, May 1989, pp. 71–76.
[44] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Database Syst.*, vol. 3, no. 3, pp. 223–247, Sept. 1978.
[45] J. M. Smith and P. Y. T. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. ACM*, vol. 18, no. 10, pp. 568–579, Oct. 1975.
[46] M. Stonebraker *et al.*, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, Sept. 1976.
[47] M. Stonebraker and L. A. Rowe, "The design of POSTGRES," in *Proc. ACM SIGMOD*, Washington, May 1986, pp. 340–355.
[48] C. Turbyfill, "Comparative benchmarking of relational database systems," PhD dissertation, Univ. of Cornell, Jan. 1988.
[49] R. E. Wagner, "Indexing design considerations," *IBM Syst. J.*, no. 4, 1973.
[50] J. Widom and S. J. Finkelstein, "A syntax and semantics for set-oriented production rules in relational database systems," *SIGMOD Rec.*, vol. 18, no. 3, pp. 36–45, Sept. 1989, Also available as IBM Res. Rep. RJ6880, June 1989.
[51] R. Williams, D. Daniels, L. M. Haas, G. Lapis, B. G. Lindsay, P. Ng, R. Obermarck, P. G. Selinger, A. Walker, P. Wilms, and R. A. Yost, "R*: An overview of the architecture," in *Proc. 2nd Int. Conf. Databases: Improving Usability and Responsiveness*, Jerusalem, June 1982. Published in *Improving Usability and Responsiveness*, P. Scheuermann, Ed. New York: Academic, pp. 1–27. Also available as IBM Res. Rep. RJ3325, Dec. 1981.
[52] P. Wilms, P. Schwarz, H. Schek, and L. Haas, "Incorporating data types in an extensible database architecture," in *Proc. 3rd Int. Conf. Data Knowledge Bases*, Jerusalem, June 1988.
[53] S. B. Yao, "Optimization of query evaluation algorithms," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 133–155, June 1979.
[54] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan, "Magic conditions," in *Proc. Principles Database Syst.*, Apr. 1990.

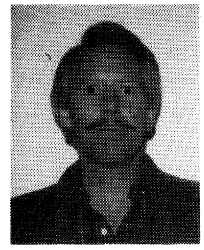


Laura M. Haas is manager of the Exploratory Database Systems Department at IBM's Almaden Research Center in San Jose. She was co-manager of the Starburst project from its inception through November 1989. She joined IBM in 1981 as a research staff member on the R* distributed relational database management project. She worked on the design and implementation of many aspects of the R* system, including catalog management, query processing, and distributed execution protocols. Before joining IBM, she studied applied mathematics and computer sciences at Harvard University, and computer science at the University of Texas at Austin, where she received the PhD degree in 1981. Her dissertation explored the problem of deadlocks in distributed systems.

Dr. Haas is a member of the Association for Computing Machinery, an Associate Editor of the *ACM Transactions on Database Systems*, and a vice-chair of SIGMOD.

Walter Chang received the B.S. degree in electrical engineering and computer science from the University of California, Berkeley, and the M.S. degree in computer science from Santa Clara University.

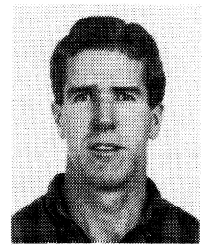
He joined IBM Research Division in 1984 and worked on extensible systems in the Starburst project. He is currently a senior member of the Technical Staff at Oracle Corporation in the Database R&D Tools Group. His main interests include object-oriented systems and multimedia databases.



Guy M. Lohman received the M.S. degree in 1975 and the Ph.D. degree in 1977 in operations research from Cornell University, Ithaca, NY, and the B.A. degree in mathematics in 1971 from Pomona College, Claremont, CA.

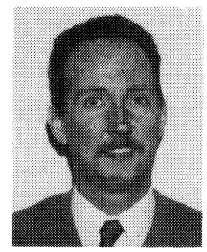
He is manager of Relational Support for Advanced Applications at IBM's Almaden Research Center in San Jose, CA, and is responsible for the rule-based query optimizer of Starburst, an extensible relational database management system prototype. Prior to that he developed, implemented, and validated the query optimizer of R*, IBM's first distributed database management system. From 1976 to 1982, he was at Caltech's Jet Propulsion Laboratory, where he founded and managed a group that developed applications of generalized DBMS and optical video disk technology to NASA's remotely-sensed geophysical databases and to DOD's distributed command, control, and communication databases. His research interests include query optimization, extensible and distributed database systems, rules systems, performance evaluation, and physical database design.

Dr. Lohman is a member of the Association for Computing Machinery, Phi Beta Kappa, Sigma Xi, Pi Mu Epsilon, and was awarded a Cornell Fellowship in 1975–1976.



John McPherson (S'75–M'81) received the B.S. degree in mathematics from the University of Michigan, Ann Arbor, and the M.S. degree in electrical engineering and the Ph.D. degree in electrical engineering and computer sciences from the University of Wisconsin, Madison.

He has been with the IBM Almaden Research Center, San Jose, CA, as a Research Staff Member since 1981. His main research activities have been in various areas of fault-tolerant computing, high-performance communications, and database management systems. He recently became a manager of the Starburst Database project at Almaden Research Center, where he has been active in the area of buffer pool management, record management, query processing, extensibility, and parallelism.



Paul F. Wilms graduated in 1976 from the University of Louvain, Belgium, as engineer in Applied Mathematics. He received in 1979 the Ph.D. degree in computer science from the National Polytechnic Institute of Grenoble, France.

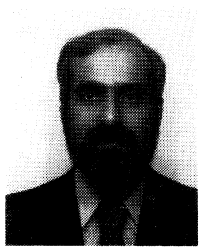
In 1980, he joined IBM Research in San Jose, and has participated since then in the design and implementation of two prototypes: R*, a distributed DBMS, and Starburst, an extensible DBMS. In 1986, he was on sabbatical in Paris teaching at Ecole Normale Superieure des Mines de Paris. His interests cover a large spectrum of database problems: distribution and replication of data, access control, incorporation of abstract data types. He has taught several courses on databases, and has lectured in different universities in Europe and the United States.

George Lapis received the B.A. degree in mathematics, from Edinboro State College in 1978 and the M.S. degree in computer science from S.U.N.Y. Binghamton in 1981.

He joined IBM in 1978 and was a member of SQL/DS development team until 1982. From that time he was working in San Jose Research in R* and Starburst projects. His research interests include user-defined functions and procedures in databases, data definition language, and authorization.

Bruce Lindsay received the B.A. degree in mathematics and the M.A. and Ph.D. degrees in computer science from the University of California at Berkeley.

He joined IBM in 1977 as a Research Staff Member where he has worked in the area of distributed data base systems. He participated in the later stages of the System R project and led the design and implementation of the R* Distributed Relational Database Prototype. Currently, he is a member of the Starburst project which is investigating extensibility mechanisms and facilities for relational database systems. His interests in database systems extend from storage, recovery, concurrency control, and operating system facilities to query languages, query analysis and execution, security, and distribution.



Hamid Pirahesh received the B.S.E.E. degree in 1972, and the M.S. and Ph.D. degrees, both in computer science from University of California, Los Angeles, in 1978 and 1983, respectively.

He has been with IBM Almaden Research Center, San Jose, CA, as a Research Staff Member since February 1985. His main activities have been in various areas of database management systems, including object-oriented support, query semantic modeling and optimization, DBMS extensibility, parallelism, and concurrency control and recovery.

Michael J. Carey received the B.S. degree in electrical engineering and mathematics and the M.S. degree in electrical engineering (computer engineering) from Carnegie-Mellon University, Pittsburgh, PA, in 1979 and 1981, respectively. He received the Ph.D. degree in computer science from the University of California, Berkeley, in 1983.

Since then he has been on the Computer Sciences Department faculty at the University of Wisconsin, Madison, where he is currently an Associate Professor. He spent the summer of 1989 as a Visiting Scientist at IBM Almaden Research Center. His research interests include concurrency control performance, distributed databases, extensible and object-oriented database systems, and database support for real-time environments. He is a co-principal investigator of the EXODUS extensible DBMS project.

Dr. Carey received an IBM Faculty Development Award in 1984, an Incentives for Excellence Award from Digital Equipment Corporation in 1986, and an NSF Presidential Young Investigator Award in 1987. He is a member of the Association for Computing Machinery, an Associate Editor of *IEEE Data Engineering*, and Secretary/Treasurer of the ACM SIGMOD group.

Eugene Shekita received the B.S. degree in computer science and electrical engineering from the University of Vermont in 1981.

He then worked for three years at Bell Laboratories in New Jersey. He is currently working towards the Ph.D. degree in computer science at the University of Wisconsin.