UTCS Technical Report 92-47
December 1992

Database Challenge: Single Schema Database Management Systems[1]

Don Batory, Vivek Singhal, Jeff Thomas
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

Many data-intensive applications require high-performance data management facilities, but utilize only a small fraction of the power of a general-purpose database system (DBMS). We believe *single schema database systems* (SSDs), i.e., special-purpose DBMSs that are designed for a single schema and a predeclared set of database operations, are a vital need of today's software industry. The challenge is to create a technology for economically building high-performance SSDs. SSD research will combine results from object-oriented databases, persistent object stores, module interconnection languages, rule-based optimizers, open-architecture systems, extensible databases, and generic data types.

1 Introduction

There are many common data-intensive applications that do not use general-purpose database systems. Consider the lock manager of a database system (DBMS). It maintains a set of tables whose structures resemble that of relations. Each table stores a different number of "tuples" (e.g., lock names and lock requests [Gra92a]); operations on tables (e.g., to set or remove locks) are atomic; and in the case of locks for long transactions, the tables are persistent and recoverable. Multi-table queries (such as retrieving identifiers of transactions that are unblocked by an unlock operation) arise frequently. Because performance is critical, the processing of queries is highly optimized. Clearly, a lock manager could use a general-purpose DBMS to manage its data, yet DBMSs are not used because of their inadequate performance and unnecessary generality.

Operating systems provide another example. Page tables, segment tables, and process tables contain "tuples" (e.g., page-table entries, segment entries, and process control blocks) that are interrelated. Multi-table queries are common (e.g., return the identifiers of pages that were referenced by process P in the last time interval) and operations in a multiprocessor environment must be atomic. Although these tables are generally neither persistent nor recoverable, the basic features that are needed to maintain page, segment, and process data are offered by general-purpose DBMSs. Once again, performance precludes DBMS usage.

There are many other similar examples: compilers query and update symbol tables, network managers access and modify routing tables, name servers maintain a horizontally partitioned and distributed table of service names and network addresses; mail systems query address alias tables (such as `.mailrc`), and persistent object stores maintain page translation tables to map virtual memory page slots to pages on disk. Essentially, any application fits this paradigm if it has special software for updating and querying persistent or nonpersistent data structures and does not use conventional database systems or object stores.

1. This research was supported in part by grants from The University of Texas Applied Research Labs, Schlumberger, Digital Equipment Corporation, and Texas Instruments.

We argue that database systems and object stores are indeed used by compilers, operating systems, network software, and database systems for the tasks noted above. It just so happens that these database systems and object stores, unlike conventional systems, were *never* designed to support a general class of schemas or ad hoc queries. Rather, they are *single-schema database systems* (SSDs), DBMSs that were designed to support a single, specific schema and a predeclared set of retrieval and update operations. Compared to general-purpose DBMSs and general-purpose object stores, SSDs provide considerably less functionality with a concomitant substantial increase in performance.

The above examples show that SSDs are pervasive in today's software. Because there are no formalizations, tools, or architectural support, SSDs are hand-crafted and are very expensive to build. There is an opportunity for database researchers to significantly improve this situation. The database community has solved very general problems of reliable and efficient data management; the challenge is to focus these general solutions to very specific situations in order to create a technology for constructing high-performance SSDs quickly and cheaply. Achieving this ambitious goal will require a synthesis and synergy of leading-edge results from databases, software engineering, programming languages, and compilers.

2 Research Challenges

It is evident from our examples that SSDs have widely varying interfaces. To make the problem of SSD generation tractable, we follow the lead of object-oriented database research. That is, a data model is needed to give SSDs a standardized interface. We then rely on object-oriented languages to allow programmers to further refine the interface for their specific needs.

As an example, stacks and queues are elementary SSDs. Clearly, stacks and queues are not relations, but sequences of tuples. SSDs generally deal with tuple sequences, which we will call *containers*. A data model for containers would define a programming interface that would be similar to the programming interfaces of today's object-oriented database systems, embedded relational languages, and persistent object stores [Cat91]. For example, cursors would be the primary means by which container objects are manipulated, and object retrievals would be declaratively specified and subject to optimization.

To bridge the gap between database-like operations on containers and the desired application-specific operations (e.g., **`push`** and **`pop`** for stacks), an abstract data type (ADT) veneer is placed over a container and each ADT operator is defined in terms of container operations [Sch78]. For example, operations on a queue **`Q`** and element **`e`** have straightforward definitions:

```
enqueue(Q,e)
{
  insert_at_end(Q,e);
}
```

```
dequeue(Q,e)
{
  cursor<Q> c;
  goto_first_obj(c);
  copy_object(c,e);
  delete(c);
}
```

```
is_empty(Q)
{
  return(is_empty(Q));
}
```

Note that container operations reflect the sequencing of container elements: the placement of inserted elements can be optionally specified at the start or end of a sequence (e.g., **`insert_at_end`**) and retrievals can begin at either end of a sequence (e.g., **`goto_first_obj`**). And just as interfaces for queues can be defined, so too can customized interfaces for more complex SSDs (e.g., lock managers) be written in terms of container operations.

The benefits of standardized SSD interfaces are obvious. First, programmer productivity is increased because the high-level abstractions provided by containers make application programs significantly easier

to write and debug. Second, application programs are now data structure (or container implementation) independent. Any container that implements a sequence without imposing a key ordering of elements (e.g., unsorted singly-linked list, unsorted doubly-linked list, circular array[2]) can be "plugged into" the above specifications to implement a queue. This significantly simplifies tuning because drastic changes to container implementations can be tried without impacting program correctness.

Given this context of SSDs as containers with user-customizable interfaces, we believe the core research problems of SSDs are:

- How does one specify an SSD implementation and how are distinctions among competing implementations expressed? This is not a problem specific to SSDs, but a fundamental problem of software engineering. (Section 3)
- Because many abstract data types are both SSDs and generics, what is the relationship between SSDs and generics? This is a problem of type systems modeling and programming language support for defining SSD implementations. (Section 4)
- Since performance is critical, how can efficient code for SSDs be generated automatically? This is a problem of applying compilation techniques to SSD program generation. (Section 5)

3 Open Architecture Systems

Disparate functionality and high performance demands encourage a wide spectrum of possible SSD features and implementations. SSDs may employ different algorithms for features such as versioning, indexing, centralized or distributed storage management, nonpersistent or persistent object storage, object level or page level concurrency control, recovery, file/data structures, and inheritance. Any or all of these features may be present in an SSD. How does one specify the particular set of choices that defines a target system and that yields a unique implementation? Research in software engineering suggests that open system architectures is a possible solution to this difficult problem [Mac90, Bat92].

Domain modeling, the process of designing open architectures, is at the forefront of software engineering research [Pri91, Ara92]. The distinguishing feature of an open architecture is a framework for defining a *family* of related systems through the use of standardized interfaces. New components can be introduced as long as they provide the same interfaces as existing system modules. A target system is an instantiation of an open architecture. By omitting optional components, it is possible to reduce (and thus tailor) the capabilities of a target system, with the potential of enhancing the system's performance. SSD technology needs precisely this capability.

The benefits of open architecture designs are well-known to the database community. TI's Open-Architecture object-oriented DBMS is a prime example [Wel92]. Extensible database systems (e.g., Postgres [Sto91], Exodus [Ric87], Starburst [Loh91], and Genesis [Bat88]) by definition have open architectures. That is, extensible DBMSs were designed to accommodate certain anticipated enhancements, such as new storage structures, new data types, and rules. SSD research can benefit from this work.

As most software design methodologies are focused on creating one-of-a-kind systems, designing open architectures is not a well-understood process. Much more research and experience is needed to advance the state of the art. Database systems, and SSDs in particular, offer a rich domain in which to do such work.

2. There are, of course, some minimal constraints. For example, circular arrays cannot be used if the maximum size of a queue exceeds the predeclared array size.

4 SSDs and Generics

SSDs and generics can be related in two different ways. One is the familiar connection of generics and data models [Atk87]. A more important, but less understood way to equate SSD components with generics. Each is considered in turn.

4.1 SSD Components as Generics

Stacks and queues are classical examples of parameterized types or *generics* [Ghe82, Gog84]. Generics have data types as parameters. For example, instantiating a stack generic with a type yields a software module for creating stacks of items of that type.

By the same reasoning, a DBMS is also generic: it is parameterized by a schema. Instantiating a DBMS with a schema yields a software system that stores and retrieves data that conforms to that schema. SSD components, which would be used in open architectures to instantiate SSD implementations, also have data types or schemas as parameters. Instantiating SSD components would yield software modules that are customized for specific data types or schemas.

We assert that generics, DBMSs, and SSD components belong to the spectrum of polymorphic types. There is clearly a tremendous gap between conventional generics (e.g., templates in C++ [Str91]) and DBMSs; generics offered by programming languages are much too simplistic to match DBMS capabilities. For example, schemas are not single record types, but large sets of interrelated record types. Generics don't offer nonprocedural query specifications and query optimization (though they should to determine the best way to search generic data structures!); rather, typical generics promote rudimentary navigational interfaces where the burden of query formulation and optimization is placed on users.

There are significant opportunities for both the database and programming language communities to advance the state of the art on generics. DBMSs, and SSDs in particular, offer rich examples to guide the theoretical and applied research efforts that will be needed. In the following sections, we suggest how research on generics and libraries of generics can benefit from SSDs. We also identify issues to be explored in database programming languages.

4.1.1 Open Architectures and Plug-Compatible Generics

Historically, interchangeable components have been very important for tuning databases. For example, Ingres allows users to select implementations for base relations and indexes from six choices (hash, heap, isam, compressed-hash, compressed-heap, and compressed-isam) [Sto76]. Each alternative is a generic data type. That is, each file structure presents exactly the same interface (which makes them plug-compatible) and is parameterized by the data type (base relation or index) to be stored.

Of the generic libraries that are available today (e.g., [Boo87, Fon90]), few generics provide the same interface. Consequently there is little or no plug-compatibility among functionally similar generics. For example, the Booch parts library defines different interfaces for binary trees and lists [Boo87]. Modifying an application that uses binary trees to one that uses lists is not always trivial and can be time-consuming. One could easily imagine (and want!) lists and binary trees as interchangeable main-memory implementations of relations. The deficiency of contemporary generic design techniques is that they do not consider plug-compatibility and open architectures, and therefore yield libraries that are unsuitable for SSDs. By the same token, generic libraries would benefit if generics which implemented the same abstraction were given the same interface.

4.1.2 Database Programming Languages

It is well-known that the polymorphism features of traditional programming languages are inadequate for writing common database applications [Oho89]; instead, sophisticated parameterizations of functions and components are required. Database programming language research has focused on introducing new generic and programming constructs to make database programming easier [Atk87]. We foresee two opportunities for research enroute to unification of generic data types and SSD components: support for attribute parameters and module expressions.

Beyond the usual type parameters, SSD/DBMS components need parameters for passing tuning constants, design options (i.e., parameters that indicate whether predefined features are to be included at component instantiation), functions and expressions (i.e., comparison functions for key fields), structure members (i.e., attributes), and imported components (i.e., what components are to provide needed lower-level services). All but the latter two parameterizations can be found in traditional polymorphic languages.

In implementations of OODBMSs, object stores, and SSDs, it is common for polymorphic functions to have structure members as parameters (e.g., function **`f(o:object,m:member)`** returns the value of member **`m`** of object **`o`**). To our knowledge, the only statically-typed polymorphic language that permits such functions to be defined is Machiavelli [Oho89], which is an extension of the functional language ML [Har88]. However, much remains to be done. It is an unsolved problem how to extend imperative programming languages (e.g., C++) and how to implement their compilers.

Open architecture designs will also impact database programming languages. As mentioned earlier, an instance of an open architecture is a composition of components. If SSD components are defined as generics in a programming language, one should be able to define their compositions in the same language. Borrowing ideas from module interconnection languages may be the answer. Broadly stated, a *module interconnection language* (MIL) is a language for specifying compositions of components [Pri86, Gog84, Pur91]. Most MILs are aimed at using components that are written in different languages and that are located on different machines to give the appearance of a uniform-language, single-site computing environment. While we do not claim that MILs are the most appropriate language for SSDs, the basic features of a MIL — namely, defining compositions of components — will indeed be necessary.

4.2 Data Models and Data Languages

SSDs will need a formal data model and algebra of containers. This model should equate schemas with types (as we have been advocating). Instances of schemas, which are databases, would become first class objects. Herein lies a wealth of research opportunities.

Consider the nesting of generics: it is common for programmers to create data structures that are, for example, lists of binary trees. An SSD counterpart to nested generics might be a container of containers or a database of databases. Functions could then be defined to return databases as their output. For example, it is common in scientific applications to encounter functions that transform a complex data set (i.e., a database of schema S1) into another complex data set (i.e., a database of schema S2) during exploratory analysis [Wil89, Min92]. Raising the level of abstraction beyond individual relations and classes would enable SSD users to model their applications more naturally. Work on complex objects [Kel91] and nested relations [Kor91] would be relevant here.

Nested generics suggests other problems. How would queries on nested databases be specified? How would such queries be optimized? Consider a container of databases. Each database would be a different instance of the same schema, and would have different object populations. Optimizing a query for one database is straightforward. Optimizing a query over *all* databases is more problematic; what might be an

efficient access plan for one database may be inefficient for another. Dynamic query evaluation techniques might be appropriate [Gra89].

5 SSD Compilers

The primary tool for creating SSDs will be compilers, in which SSD components are coded and their compositions are defined. Like traditional compilers, an SSD compiler should strive to generate code of comparable or better efficiency than that written by the average programmer. Inefficient code for managing DBMS lock tables, OS process tables, and compiler symbol tables cannot be tolerated. Achieving very high efficiencies poses a challenge to the database and compiler communities.

The strategy used for realizing C++ templates might be appropriate: templates are macro expanded and partially evaluated when parameters are instantiated [Str91]. The result is a software module that is both customized and optimized for the given application; the generality of the template has given way to an efficient and specialized instantiation. By analogy, an SSD compiler would "expand" and "simplify" general-purpose DBMS code (or rather, a composition of SSD components) for a specific schema, resulting in an SSD that is customized and optimized for that schema and its set of predeclared operations. In effect, the generality of a DBMS has given way to an efficient and specialized instantiation.

To see what is needed to "expand" and "simplify" a composition of SSD components, consider today's object-oriented DBMSs. They are able to process declarative queries and high-level updates that are specified in application programs, and to generate efficient code for their fixed implementations [Cat91]. SSDs will require a much more general technology because the code for compiled queries and updates must be retargetable to any member of a large class of implementations.

SSD code generation will require a melding of coarse grain optimizations, like those of today's query optimizers, with fine grain optimizations of today's compilers. Work on automatic programming, such as AP5 [Coh91], and compilers for very high level languages, such as SETL [Fre83], may be relevant. Another possibility is to apply partial evaluation techniques of compilers to intertwine and optimize "modular" specifications to produce an efficient monolithic implementation of a target subsystem [Fre89]. Domain-specific compilers for open architecture systems are also promising. These compilers exploit knowledge and optimization techniques that are specific to a domain to generate efficient code. The Morpheus network protocol compiler and the Predator data structure compiler are prime examples [Abb92, Sir92, Bat92].

5.1 SSD Query Optimizers

SSDs will need modular implementations of query optimizers. Research on rule-based optimizers (e.g., Volcano [Gra92b] and Starburst [Haa89]) is relevant because of the inherent flexibility of these optimizers to admit new operators and algorithms in determining efficient strategies for processing queries.

Optimizing queries for main-memory containers (i.e., data structures) poses a problem when statistics (that are needed for optimization) are not available. It is possible to instrument SSD implementations (using `gprof`-like techniques) to gather such statistics. Two different approaches to optimize queries can then be taken. Multiple access plans could be generated by an SSD optimizer/compiler. At run-time, the best available plan would be chosen based on available statistics [Gra89]. In this way, the target SSD would automatically readjust itself to a changing environment. Alternatively, the statistics could be used to recompile and reoptimize the SSD, thus avoiding the overhead of evaluating plans dynamically. This latter approach belongs to a much larger subject area: self-tuning SSDs.

5.2 Self Tuning SSDs

Open architectures with component libraries make two things possible: (1) an SSD can be defined as a legal composition of components and (2) the domain of all constructable SSDs is the set of all such compositions. Because libraries have finite numbers of components, the domain of SSD implementations are *enumerable*, which makes possible unprecedented opportunities for automatically optimizing database/SSD designs.

The idea of self-tuning software (e.g., databases) has long been of interest to the database community. The idea is simple: databases or SSDs periodically record information about their workload. By using this information to guide a search through the domain of SSD implementations (i.e., space of legal SSD component compositions), SSDs would reconfigure themselves periodically for better performance. Note that this is a *much* broader problem than optimizing individual queries: it is the storage structures *and* access plans by which data is to be stored and retrieved that are being optimized.

Searching the space of SSD implementations is a daunting task. I/O costs are estimated by the leaf (or lowest-level) components of an SSD, because that is where interactions with operating systems occur. This means, unfortunately, that cost formula cannot be evaluated fully until a complete SSD implementation has been specified. It is not at all obvious how to search this space efficiently. Furthermore, the computational complexity of optimizing a single composition can be NP-hard. Early research on database optimization problems may be relevant here [Mit75].

It is worth noting that *profile-based code optimization* (i.e., self-tuning programs) is an active subject of research in the compiler community [Cha91]. Research on self-tuning SSDs would be an extension to this line of research.

6 Challenge Summary

There are many common data-intensive applications that cannot use general-purpose DBMSs or general-purpose object stores because of unnecessary generality, inadequate performance, or lack of specific features. These applications have instead relied on highly-optimized hand-crafted data storage systems, which we call single-schema database systems (SSDs). Because there are no tools, formalizations, or architectural support, SSDs are expensive to build.

The database community has solved very complex and general problems in data management. The challenge is to apply these general solutions to a specific application to produce highly-efficient SSDs in a cost-effective manner. Realizing SSD compilers will force researchers in the database, software engineering, programming language, and compiler communities to address fundamental problems and to relate their work to corresponding efforts that are already underway in neighboring communities.

Acknowledgments. We are grateful for the detailed comments and helpful suggestions for improvement from Ira Baxter, Dinesh Das, Curtis Dyreson, Christoph Freytag, Mark Johnstone, Sheetal Kakkad, Nick Kline, Marty Sirkin, Avi Silberschatz, Rick Snodgrass, Michael Soo, Nandit Soparkar, and Craig Thompson on an earlier draft of this paper.

7 References

[Abb92] M. B. Abbott and L. L. Peterson, "A Language-Based Approach to Protocol Implementation", Tech. Rep. 92-2, Department of Computer Science, University of Arizona, 1992.

[Ara92] G. Arango and B. Blum, editors, "Special Issue: Applications of Domain Modeling to Software Construction", *International Journal of Software Engineering and Knowledge Engineering*, Sept. 1992.

[Atk87] M. P. Atkinson and O. P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, June 1987.

[Bat88] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise, "GENESIS: An Extensible Database Management System", *IEEE Trans. Software Engr.*, November 1988, 1711-1730.

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems With Reusable Components", *ACM TOSEM*, Oct. 1992.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin Cummings, 1987.

[Car85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, December 1985, 471-522.

[Cat91] R. G. G. Cattell, editor, "Special Issue: Next Generation Database Systems", *Communications of the ACM*, October 1991.

[Cha91] P. P. Chang, S. A. Mahlke, and W-M. W. Hwu, "Using Profile Information to Assist Classical Code Optimizations", *Software-Practice and Experience*, December 1991, 1301-1321.

[Coh91] D. Cohen, AP5 Manual, USC Information Sciences Institute, 1991.

[Fon90] M. Fontana, L. Oren, and M. Neath, "COOL – A C++ Object-Oriented Library", Texas Instruments Inc., Computer Science Center, Dallas, Texas, 1990.

[Fre83] S. M. Freudenberger, J. T. Schwartz, and M. Sharir, "Experience with the SETL Optimizer", *ACM TOPLAS*, March 1983.

[Fre89] J. C. Freytag and N. Goodman, "On the Translation of Relational Queries into Iterative Programs", *ACM TODS*, March 1989, 1-27.

[Ghe82] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, 1982.

[Gog84] J. Goguen, 'Parameterized Programming', *IEEE Transactions on Software Engineering*, Sept. 1984.

[Gra89] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans", *ACM SIGMOD* 1989, 358-366.

[Gra92a] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufman, 1992.

[Gra92b] G. Graefe and W. McKenna, "Extensibility and Search Efficiency in the Volcano Optimizer Generator", Tech. Rep. CU-CS-91-563, Computer Science Department, University of Colorado at Boulder, 1992.

[Haa89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, "Extensible Query Processing in Starburst", *ACM SIGMOD* 1989, 377-388.

[Har88] R. Harper, R. Milner, and M. Tofte, "The Definition of Standard ML (Version 2)", Tech. Rep. ECS-LFCS 88-62, Department of Computer Science, University of Edinburgh, August 1988.

[Kel91] T. Keller, G. Graefe, and D. Maier, "Efficient Assembly of Complex Objects", *ACM SIGMOD 91*, 148-157.

[Kor91] H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1991.

[Loh91] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. Bernhard Schiefer, “Extensions to Starburst: Objects, Types, Functions, and Rules”, in [Cat91], 94-109.

[Mac90] D. MacKinnon, W. McCrum, and D. Sheppard, *An Introduction to Open Systems Interconnection*, Computer Science Press, New York, 1990.

[Min92] W. H. Miner, private correspondence, Fusion Research Center and Institute for Fusion Studies, The University of Texas at Austin, 1992.

[Mit75] M. F. Mitoma and K.B. Irani, “Automatic Database Schema Design and Optimization”, *VLDB 1975*, 286-321.

[Nov92] G. Novak, “Software Reuse by Compilation through View Clusters”, Dept. Computer Sciences, University of Texas at Austin, 1992.

[Oho89] A. Ohori, P. Buneman, and V. Breazu-Tannen, “Database Programming in Machiavelli – A Polymorphic Language with Static Type Inference”, *ACM SIGMOD* 1989, 46-57.

[Pri86] R. Prieto-Diaz and J. M. Neighbors, “Module Interconnection Languages”, *Journal of Systems and Software*, Nov. 1986, 307-334. Also in P. Freeman, editor, *Software Reusability*, IEEE Computer Society Press, Los Alamitos, CA, 1987.

[Pri91] R. Prieto-Diaz and G. Arango, editors, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Pur91] J. R. Purtilo, R. Snodgrass, and A. L. Wolf, “Software Bus Organization: Reference Model and Comparison of Two Existing Systems”, Module Interconnection Formalism Technical Note No. 8, November 1991.

[Ric87] J. E. Richardson and M. J. Carey, “Programming Constructs for the Database System Implementation in EXODUS”, *ACM SIGMOD* 1987, 208-219.

[Sch78] J. Schmidt, “Some High Level Language Constructs for Data of Type Relation”, *ACM TODS*, 1978.

[Sir92] M. Sirkin, D. Batory, and V. Singhal, “Software Components in a Data Structure Precompiler”, TR-92-29, Dept. Computer Sciences, University of Texas at Austin, May 1992.

[Sto76] M. Stonebraker, E. Wong, P. Kreps, and G. Held, ‘The Design and Implementation of INGRES’, *ACM TODS*, 1 #3 (Sept. 1976), 189-222.

[Sto91] M. Stonebraker and G. Kemnitz, “The POSTGRES Next-Generation Database Management System”, in [Cat91], 78-93.

[Str91] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, 1992.

[Wel92] D. L. Wells, J. A. Blakeley, and C. W. Thompson, “Architecture of an Open Object-Oriented Database Management System”, *Computer*, Vol. 25, No. 10, Oct. 1992.

[Wil89] J. C. Wiley, et al., “The TEXT data base”, *Rev. Sci. Instrum*., 60 (7), July 1989.