Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System

Surajit Chaudhuri
Microsoft Research
Redmond, WA 98052, USA
surajitc@microsoft.com

Gerhard Weikum
University of the Saarland
D-66123 Saarbruecken, Germany
weikum@cs.uni-sb.de

Abstract

Database technology is one of the cornerstones for the new millennium's IT landscape. However, database systems as a unit of code packaging and deployment are at a crossroad: commercial systems have been adding features for a long time and have now reached complexity that makes them a difficult choice, in terms of their "gain/pain ratio", as a central platform for value-added information services such as ERP or e-commerce. It is critical that database systems be easy to manage, predictable in their performance characteristics, and ultimately self-tuning. For this elusive goal, RISC-style simplification of server functionality and interfaces is absolutely crucial. We suggest a radical architectural departure in which database technology is packaged into much smaller RISC-style data managers with lean, specialized APIs, and with built-in self-assessment and auto-tuning capabilities

1. The Need for a New Departure

Database technology has an extremely successful track record as a backbone of information technology (IT) throughout the last three decades. High-level declarative query languages like SQL and atomic transactions are key assets in the cost-effective development and maintenance of information systems. Furthermore, database technology continues to play a major role in the trends of our modern cyberspace society with applications ranging from web-based applications/services, and digital libraries to information mining on business as well as scientific data. Thus, *database technology* has impressively proven its benefits and *seems* to remain crucially relevant in the new millennium as well.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

Success is a lousy teacher (to paraphrase Bill Gates), and therefore we should not conclude that the *database system*, as the unit of engineering, deploying, and operating packaged database technology, is in good shape. A closer look at some important application areas and major trends in the software industry strongly indicates that database systems have an overly low "gain/pain ratio". First, with the dramatic drop of hardware and software prices, the expenses due to human administration and tuning staff dominate the cost of ownership for a database system. The complexity and cost of these feed-and-care tasks is likely to prohibit database systems from further playing their traditionally prominent role in the future IT infrastructure. Next, database technology is more likely to be adopted in unbundled and dispersed form within higher-level application services.

Both of the above problems stem from packaging all database technology into a single unit of development, maintenance, deployment, and operation. We argue that this architecture is no longer appropriate for the new age of cyberspace applications. The alternative approach that we envision and advocate in this paper is to provide RISC-style, functionally restricted, specialized data managers that have a narrow interface as well as a smaller footprint and are more amenable to automatic tuning.

The rest of the paper is organized as follows. Section 2 puts together some important observations indicating that database systems in their traditional form are in crisis. Section 3 briefly reviews earlier attempts for a new architectural departure along the lines of the current paper, and discusses why they did not catch on. Section 4 outlines the envisioned architecture with emphasis on RISC-style simplification of data-management components and consequences for the viability of auto-tuning. Section 5 outlines a possible research agenda towards our vision.

2. Crisis Indicators

To begin our analysis, let us put together a few important observations on how database systems are perceived by customers, vendors, and the research community.

Observation 1: Featurism drives products beyond manageability. Database systems offer more and more features, leading to extremely broad and thus complex

interfaces. Quite often novel features are more a marketing issue rather than a real application need or technological advance; for example, a database system vendor may decide to support a fancy type of join or spatial index in the next product release because the major competitors have already announced this feature. As a result, database systems become overloaded with functionality, increasing the complexity of maintaining the system's code base as well as installing and managing the system. The irony of this trend lies in the fact that each individual customer (e.g., a small enterprise) only makes use of a tiny fraction of the system's features and many high-end features are hardly ever exercised at all.

Observation 2: SQL is painful. A big headache that comes with a database system is the SQL language. It is the union of all conceivable features (many of which are rarely used or should be discouraged to use anyway) and is way too complex for the typical application developer. Its core, say selection-projection-join queries and aggregation, is extremely useful, but we doubt that there is wide and wise use of all the bells and whistles. Understanding semantics of SQL (not even of SQL-92), covering all combinations of nested (and correlated) subqueries, null values, triggers, ADT functions, etc. is a nightmare. Teaching SQL typically focuses on the core, and leaves the featurism as a "learning-on-the-job" life experience. Some trade magazines occasionally pose SQL quizzes where the challenge is to express a complicated information request in a single SQL statement. Those statements run over several pages, and are hardly comprehensible. When programmers adopt this style in real applications and given the inherent difficulty of debugging a very high-level "declarative" statement, it is extremely hard if not impossible to gain high confidence that the query is correct in capturing the users' information needs. In fact, good SQL programming in many cases decomposes complex requests into a sequence of simpler SQL statements.

Observation 3: Performance is unpredictable. Commercial database engines are among the most sophisticated pieces of software that have ever been built in the history of computer technology. Furthermore, as product releases have been driven by the time-to-market pressure for quite a few years, these systems have little leeway for redesigning major components so that adding features and enhancements usually increases the code size and complexity and, ultimately, the general "software entropy" of the system. The scary consequence is that database systems become inherently unpredictable in their exact behavior and, especially, performance. Individual components like query optimizers may already have crossed the critical complexity barrier. There is probably no single person in the world who fully understands all subtleties of the complex interplay of rewrite rules, approximate cost models, and search-space traversal heuristics that underlie the optimization of complex queries. Contrast this dilemma with the emerging need for performance and service quality guarantees in e-commerce, digital libraries, and other Internet applications. The PTAC report has rightly emphasized: "our ability to analyze and predict the performance of the enormously complex software systems that lie at the core of our economy is painfully inadequate" [18].

Observation 4: Tuning is a nightmare and auto-tuning is wishful thinking at this stage. The wide diversity of applications for a given database system makes it impossible to provide universally good performance by solely having a well-engineered product. Rather all commercial database systems offer a variety of "tuning knobs" that allow the customer to adjust certain system parameters to the specific workload characteristics of the application. These knobs include index selection, data placement across parallel disks, and other aspects of physical database design, query optimizer hints, thresholds that govern the partitioning of memory or multiprogramming level in a multi-user environment. Reasonable settings for such critical parameters for a complex application often depend on the expertise and experience of highly skilled tuning gurus and/or time-consuming trial-and-error experimentation; both ways are expensive and tend to dominate the cost of ownership for a database system. "Auto-tuning" capabilities and "zero-admin" systems have been put on the research and development agenda as high priority topics for several years (see, e.g., [2]), but despite some advances on individual issues (e.g., [4,7,8,10,24]) progress on the big picture of self-tuning system *architectures* is slow and a breakthrough is not nearly in sight. Although commercial systems have admittedly improved on ease of use, many tuning knobs are merely disguised by introducing internal thresholds that still have to be carefully considered, e.g., at packaging or installation time to take into account the specific resources and the application environment. In our experience, robust, universally working default settings for complex tuning knobs are wishful thinking. Despite the common myth is that a few rules of thumb could be sufficient for most tuning concerns, with complex, highly diverse workloads whose characteristics evolve over time it is quite a nightmare to find appropriate settings for physical design and the various run-time parameters of a database server to ensure at least decent performance.

Observation 5: We are not alone in the universe. Database systems are not (or no longer) at the center of the IT universe. Mail servers, document servers, web application servers, media servers, workflow management servers, e-commerce servers, auction servers, ERP systems, etc. play an equally important role in modern cyberspace applications. A good fraction of these higher-level services have their own specialized storage engines and, to some extent, simple query engines, and even those that do make use of a database system utilize only a tiny fraction of the functionality up to the extreme point where the database system is essentially used as a luxurious BLOB manager. Thus, while we have been busy building

universal database management systems, many important applications have simply decided to go on their own. In particular, quite a few vendors of such generic applications view complex SQL and the underlying query optimizer as a burden rather than an opportunity and consequently ignore or circumvent most high-level database system features (e.g., see [17]). Of course, this approach often amounts to "re-inventing" certain query optimization techniques in the layer on top of the database systems, but the vendors of these "value-added" services consider the re-implementation as appropriate given that it can now be streamlined and tailored to a specific application class.

Observation 6: System footprint considered harmful. The market for "classical" database installations may saturate in the near future. On the other hand, significant growth potential for the database system industry lies in embedded applications such as information services in cellular phones, cars, palm-pilots, etc.. These applications need even smaller fractions of the rich feature sets offered by a full-fledged database system. On the other hand, they face much tighter resource constraints in their embedded settings: neither the customer's wallet nor the battery supply of a typical gizmo easily allows purchasing another 256 MB of memory. This implies that the footprint of a system (i.e., code-size and especially memory requirements) is a key issue for lightweight embedded applications. Also, brute-force solutions to performance problems (i.e., adding hardware) are often infeasible. Thus, the current generation of commercial database systems, despite some of them advertising lightweight versions, are not geared for these settings.

Observation 7: System-oriented database research is frustrating. For the academic research (and teaching) community the ever-increasing complexity of commercial database systems makes it very hard to position its individual research efforts in the fast-moving IT world. On one hand, some of the traditional database-engine topics have been almost beaten to death, and the community leaders aim to steer researchers away from studying these seemingly myopic issues (although some of them are still far from a truly satisfactory solution). On the other hand, the success of research is more and more measured in terms of product impact, and for an academic idea to be intriguing to product developers, major prototype implementation and extensive experimentation is often required. With commercial database system being such a highly complex target, this kind of work becomes less and less rewarding and all too often exceeds the available resources in a university environment. Thus, it is no wonder that systems-oriented database research has become so scarce outside the labs of the major vendors. The personal consequence for many academic researchers is that they turn away from database systems and start working on different topics, often in conjunction with startup companies where again the "entrance fee" is much lower than for a new database system). Similar observations can be made about teaching: teaching interface standards is boring, and teaching database system internals is often unsatisfactory because of the lack of documented knowledge or the indigestibility of the collection of tricks and hacks found in some systems.

All these observations together strongly indicate that database systems are in crisis: they have reached or even crossed a complexity barrier beyond which their lack of manageability and predictability will become so painful that information technology is likely to abandon database systems as a cornerstone of data-intensive applications. The bottom line is that database systems have become unattractive in terms of their *"gain/pain ratio"*: the gain of using a database system versus going on your own is outweighed by the pain of having to cope with overly sophisticated features and a hardly manageable, humongous piece of software.

3. Explanations and Previous Attempts

3.1 Explanations

Database systems (and possibly software technology in general) are susceptible to several "traps": principles that open up both opportunities and complexity (just like Pandora's box in the Greek mythology) and have been overstressed in the evolution of database systems.

Trap 1, the "universality trap": Since a computer is a universal tool, software developers strive for extremely versatile, general-purpose solutions. This attitude of trying to cover as much ground as possible is ambivalent and extremely delicate, however. After all, every good principle can be generalized to the point where it is no longer useful. In the case of database systems, this has led to extremely complex, and hard-to-manage software packages. Contrast this evolution with other fields of engineering: despite the fact that wheels or engines are universal components of cars, the automobile industry has not attempted to develop a universal car that unifies all features of a sports convertible, luxury limousine, 4WD, and economic as well as "ecologically correct" compact into a single product.

Trap 2, the "cost trap": Since software is inexpensive to "manufacture", i.e., copy and distribute to customers, database systems tend to agglomerate too much into a single package of deployment. This approach disregards the cost of maintenance, system operation, and especially the "cost" of gaining confidence in the dependability of the product comprising both correct behavior in all situations and predictable, ideally guaranteed performance. The latter is, of course, more a problem for customers (and also developers of value-added services on top of database systems) than for the vendors.

Trap 3, the "transparency trap": A "transparency trap" arises from very high-level features that hide execution

costs. Historically, Algol 68 is a good historic example for this kind of pitfall. This was an extremely powerful language (for the standards of that period), and programmers could easily write code that would end up being a run-time nightmare. A similar phenomenon of hidden execution cost and thus careless programming can be observed with SQL. We teach our students that they can exploit its full expressiveness and rely on the query optimizer producing a miracle, but real life is pretty different. Efficiency should no longer be the main driving force in program designs and coding, but it should not be completely disregarded either. The concepts of a computer language should steer programmers towards reasonably efficient code rather than offering features for which there is little hope that their runtime is acceptable.

Trap 4, the "resource sharing trap": By putting as much functionality as possible into a single software box with certain underlying hardware resources (disks, memory, etc.), we can dynamically share these resources for different purposes. For example, by storing videos in a relational database, video streaming can exploit the space and performance capacity of disks that hold also conventional tables and indexes during daytimes when disks are lightly used for the OLTP or OLAP part of the business. Likewise, by running business-object-style application code, in the form of user-defined functions and abstract data types (ADTs), in the database server, the server's memory and processors are dynamically shared between query and application processing. The flip side of this coin is that such resource sharing introduces interesting but horribly complex tuning problems. Disk configuration planning and disk scheduling for both video streaming and conventional data accesses are much harder than dealing with each of the two data categories separately on two disjoint disk pools. Query optimization in the presence of ADTs is an intriguing research problem, but seems like trying to solve a combined problem before we fully understand each of its constituents.

Some people may argue that we should completely give up building complex software systems, and go back to the roots of programming with every line of code documented by its mathematical properties (e.g., invariants) and not release any software package whose correctness is not rigorously verified. But we should, of course, avoid this unrealistic attitude and the "humble programmer trap", too.

3.2 Previous Attempts

We are surely not the first ones who have made similar observations, have drawn potential conclusions, and have toyed with possible departures from the beaten paths. In the following we briefly discuss some of the most prominent attempts and why we believe they did not achieve their goals.

Attempt 1, database system generators: A common approach in computer science is to address problems by going to a meta level. In the specific case of database system architecture, [3] has proposed to generate customized database systems from a large library of primitive components (see also [13] for a related generator approach). The focus of this interesting but ultimately not so successful work was on storage and index management, but even in this limited context it seems impossible to implement such a generator approach in a practically viable form. Also, once such approaches take into account also cache management, concurrency control, recovery, query optimization, etc., they would inevitably realize the many subtle but critically important interdependencies among the primitive components, which makes the generation of correct and efficient system configurations awfully difficult.

Attempt 2, extensible kernel systems: A related approach that avoids some of the pitfalls of the generator theme has been to put core functionality into a kernel system and provide means for extending the kernel's functions as well as internal mechanisms [5,15,19,23]. This approach has led to the current generation of "data blades", "cartridges", and "extenders" in object-relational products [6]. The extensibility with regard to user-defined functions and ADTs is working reasonably well, but the extensibility with regard to internals, e.g., adding a new type of spatial index, is an engineering nightmare as it comes with all sorts of hardly manageable interdependencies with concurrency control, recovery, query optimization, etc. We would even claim that functional extensibility also creates major problems with respect to managing the overall system in the application environment; especially predicting and tuning the system's performance becomes horribly complex. In fact, it seems that all data-blade-style extensions are written by the database system vendor (or some of their partners) anyway. So extensibility on a per application basis appears to remain wishful thinking, and vendor-provided extensions that are coupled with the "kernel" (actually, a full-fledged database system anyway) are just additional complexity from the customers' viewpoint.

Attempt 3, unbundled technology: More recently several senior researchers have proposed to view database technology, i.e., our know-how about storage management, transaction management, etc., as something that is independent from the packaging of this technology into the traditional form of database systems [1,12,22]. A practical consequence (not necessarily fostered by but fully in line with these positions) is that we see more and more mature, industrial-strength database technology in all sorts of non-database products such as mail servers, document servers, specialized main-memory servers for switching and billing in telecommunication. In contrast to the situation ten years ago, most of these products have state-of-the-art algorithms for recovery, caching, etc. With a broad variety of such services, building

applications that span multiple services becomes more difficult. Identifying principles for federating services into value-added information systems is subject of ongoing and future research.

The third approach, unbundling database technology and exploiting it in all sorts of services, is closest to our own position. However, the previous position papers have not said anything about the future role of database systems themselves as a package of installation and operation. We aim to redefine the scope and role of database servers, and outline also a possible path towards a world of composable "RISC"-style data management blocks.

4. Towards RISC-style Components

In computer architecture, the paradigm shift to RISC microprocessors led to a substantial simplification of the hardware-software interface. We advocate a similarly radical simplification of database system interfaces. Our proposal calls for building database systems with components that are built with RISC philosophy in mind. The components need to be carefully designed so that they enable building of richer components on top of the simpler components while maintaining a clear separation among components. Thus, each RISC-style component will have a well defined and relatively narrow functionality (and API). There is a need for "universal glue" as well to enable components to cooperate and build value-added services.

RISC data-management components appear attractive for us for three reasons. First, such components, with their relatively narrow functionality (and API), give us some new hope for predictable behavior and self-tuning capabilities. Second, the narrow functionality, coupled with the ability to build value-added services "on-top" makes such components far more attractive for varied information applications of today compared to a monolithic "universal" database management system. Third, even if we only build a monolithic database management system and no other data management services, building it using RISC data-management components will make it far more predictable and tunable than today's database management systems. All these benefits apply equally to RISC-style componentization within a database system as well as across different kinds of data managers for IT systems in the large.

RISC philosophy for database systems:

We will illustrate our intuition on RISC data components by focusing on querying capabilities of the database system. The simplest class of query engine is *a single-table selection processor*, one that supports single-table selection processing and simple updates with B+-tree indexing built-in. Such a single-table engine can be used in many simple application contexts. Indeed, with transactional support, such a query engine provides an ideal platform that can be used by many applications. In fact, until recently, even SAP R/3 essentially used the underlying DBMS as a storage engine of this kind. Another advantage of such a selection processor is that it can be used with a programmer-friendly API, with little or no knowledge of SQL. Another class of query engine that is quite attractive is the *Select-Project-Join (SPJ) query-processing engine*, suitable for OLTP and simple business applications. Such a RISC-component service can build on top of the simpler single table selection processor, much like how RDS was layered on RSS in System-R. The theory and performance of a SPJ query processor is much more clearly understood than that of a full-blown SQL query engine. In fact, the simplicity of the System-R optimizer has led to a deep understanding of how to process such queries efficiently, with join ordering, local choice of access methods, and interesting orders being the three central concepts. Adding *support for aggregation* to the SPJ engine, through sorting, data-stream partitioning (possibly hierarchical) capabilities, and more powerful local computation within a group (e.g., see [9]) brings it closer to requirements of decision support applications and specifically OLAP. Furthermore, note that the support for aggregation needs to be stronger than what SQL supports today. Yet, layering enables us to view the optimization problem for SPJ+Aggregation query engine as the problem of moving (and replicating) the partitioning and aggregation functions on top of SPJ query sub-trees. The challenge in designing such a RISC-component successfully is to identify optimization techniques that require us to enumerate only a few of all the SPJ query sub-trees. Finally, one could implement a full-fledged *SQL processor* that exploits the SPJ+Aggregation engine. Despite building the same old SQL engine, such an approach offers the hope of decomposing the optimization problem and thus the search complexity. Although in principle such a layered approach can result in inferior plans, it offers the hope of controlling the search space for each layered component much more tightly. Furthermore, the ad-hoc-ness in many of the commercial optimizer search techniques leave us far from being convinced that a global search would necessarily lead to better solutions in most cases. The above discussion illustrates how database technology may be re-packaged into layers of independently usable and manageable components. It should be noted that the vision that such "reduced-functionality" data managers will be available is implicit and anticipated in the OLE-DB API. However, provisions in the API have not led to products with such components yet.

In a similar vein, even the architecture of *storage managers* could be opened up for RISC-style redesign. A number of key mechanisms like disk management, caching, logging and recovery, and also concurrency control are undoubtedly core features in every kind of storage manager. But when we consider also the corresponding strategies that drive the run-time decisions of these mechanisms, we realize that different data or

application classes could prefer tailored algorithms, for example, for cache replacement and prefetching, rather than universal algorithms that need more explicit tuning. For example, a media manager that stores video and audio files and needs to provide data-rate guarantees for smooth playback at clients would differ from the storage manager underneath a relational selection processor. Similarly, a semi-structured data storage manager that is geared for variances in record lengths and structures may choose implementation techniques that differ from those of schematic table storage. Finally, we could argue that even an *index manager* should be separated from the primary-data storage. This would make immediate versus deferred index maintenance equal citizens, with the latter being the preferable choice for text or multimedia document management. The price for this extreme kind of specialization and separation would be in additional overhead for calls across components; in the extreme case we may need an explicit two-phase commit between the primary-data and the index storage manager. However, this price seems to be tolerable for the benefit of removing tuning options that a universal storage manager would typically have and a much better chance of automating the remaining fine-tuning issues because of the smaller variance in the per-component workload characteristics.

The "RISC"-style componentization has some important ramifications. First, such componentization limits the interactions among components. For example, the SPJ query engine must use the "selection processor" as an encapsulated engine accessed through the specific APIs. The SPJ engine will have no knowledge of the internals of the selection processor except what can be gleaned via the API published for any consumer of the selection processor. Thus, the SPJ processor will know no more or no less than any other consumer of the "single-table selection" processor. Second, the API provided by any such component must expose at least two classes of interfaces: *functionality* as well as *import/export of meta-information*. For example, in the case of a selection processor, the functionality interface enables specification of a query request (table name and a filter on a column of the table). The import/export interface can expose to external components *limited* information that determines performance of the selection processor. In particular, the selection processor (specifically, the optimizer) should support an interface to return estimated selectivity of (predicates in) a query as well as the estimated run-time and resource consumption of executing the query. Note that obtaining reasonably accurate run-time estimates is important not only for query optimizers to choose an execution plan but also for deciding what priority a query should be given or whether it is worthwhile to submit it all (an issue that frequently arises in OLAP). Conversely, through an import interface, we can empower the application to specify parameters (e.g., response-time or throughput goals) that influence query execution.

RISC philosophy for IT systems in the large:

For building IT systems in the large, we obviously need more building blocks than the various database services discussed above. So we need to apply similar considerations also to web application servers, message queue managers, text document servers, video/audio media servers etc. Each of them should be constructed according to our RISC philosophy, and all of them together would form a library of RISC-style building blocks for composing higher-level, value-added information services. We have to make sure that the complexity that we aim to reduce by simplifying the components does not reappear at the global application level. Research into principles of service composability should thus be put high on our community's agenda (a serious discussion of these issues is beyond the scope of this paper).

The challenge in adopting a RISC-style architecture is to precisely identify and standardize the functionality and import/export interfaces for each component such that the following objectives are met: (1) these interfaces can be exploited by a multitude of applications, (2) the performance loss due to the encapsulation via the API results is tolerable, and (3) each individual component is self-tuning and exhibits predictable performance. The last point is worth reemphasizing since it attempts to explain as to why RISC-style components are of great importance. It is only when we construct components that can be "locally" understood and explained, that we can hope to achieve predictability of performance and the ability to tune them as a component. As an example, we feel that it is much easier to understand the behavior (and indeed theory) of a single-table selection processor, or that of an SPJ processor that is built using only the narrow interfaces exposed by a selection processor. Of course, use of such narrow interfaces and creation of limited functionality can hamper performance, but as long as the degradation is marginal or even moderate, the need for predictable performance and self-tuning far outweighs such concerns given the cost point of today's commodity hardware. We now describe some of the important ramifications of the architecture we are espousing.

4.1 Notable Departures from Today's Architectures

The new departure that we advocate in this paper follows, in spirit, the earlier approaches of system generators and unbundling (referred to as Attempts 1 and 3 in Section 3.2). However, in contrast to these earlier proposals, we consider the appropriate packaging as a vital aspect of the overall architecture. In comparison to the modules considered by a generator, our RISC-style components are much coarser and ready for self-contained installation, thus avoiding the pitfall of the many subtle inter-dependencies among overly fine-grained modules. Similarly, we go beyond the unbundling theme by striving for components that are self-contained also in terms of

predictability and self-tuning, and we aim to reduce the complexity of components and are willing to lose some performance to this end. The following simplifications would be important steps in this direction.

Support only for limited data types: The mantra in our new architecture is predictability and auto-tuning. This strongly argues against support for arbitrary data types. Instead, database systems should focus on the data types that they are best at, namely, tables with attributes of elementary type. Essentially this means going back to the roots of database systems, business data. There is no truly compelling reason for supporting the entire plethora of application-specific and multimedia data types. The initial motivation for adding all these features has been to ease data integration. But integration does not imply universal storage. In fact, what replaces the need for universal storage are data exchange protocols, advanced APIs, and component models that enable specialized storage/query systems to federate with traditional databases, e.g., OLE-DB/COM, EJB, or emerging XML protocols. To reiterate, limiting the responsibilities of the database system to data in table format makes the system much more manageable and give us a significantly better handle on the, still remaining and all but trivial, (automated) tuning problem.

No more SQL: As mentioned in Section 2, there is no demand for much of the complexity of full-fledged SQL, nor for the ”Select-From-Where” syntactic sugar that has outlived its original motivation. We advocate a streamlined API through which programs essentially submit *operator trees* to the database server modules. In the example above, a selection processor will accept operator trees that only contain scan and selection; an SPJ processor will be able to process requests for processing trees that contain selection, projection, and join operators. For convenience, these operator trees may be linearized. Moreover, they should avoid the complex forms of nested and correlated subqueries that make current SQL hard to master. However, only changing the syntactic form of SQL is not enough. The language constructs themselves have to be simplified. Given that so many intelligent people have already argued for a simplification for the past two decades without success, we are convinced that the key for simplification lies in substantially limiting the functionality and expressiveness (which most of the previous attempts did not want to compromise).

Disjoint, manageable resources: There should be no dynamic resource sharing among components. This simplifies performance tuning and also provides additional isolation with regard to software failures. For example, when building a news-on-demand service on top of a video server, a text document server, and an SPJ table manager (for simple business issues such as accounting), each of these servers should have its own, dedicated hardware for simpler tuning (although this rules out additional cost/performance gains via dynamic resource sharing). In the extreme this could imply, for example, that, within a table manager, data pages and index pages should always reside on disjoint disks (which is a commonly used rule of thumb anyway) for the sake of simplicity and at the modest cost of a few additional disks.

Pre-configuration: Each RISC-style data management component could be pre-configured for say five or ten different “power levels” with regard to feature-richness (e.g., "basic", "standard", "advanced", and "full") as well as performance and/or availability levels. Along the latter lines one could support a small spectrum of data server models such as "good for mostly-read workloads", "good for small to medium data volumes". This pre-configuration approach promises a viable compromise between "one size fits all" and complete freedom for configuration and feature selection at the risk of facing a monstrous tuning problem. In many cases the need for customization and tuning could be completely eliminated, at least at installation times. Note that this trend towards pre-packaging can already be observed in the IT industry, definitely as far as marketing and sales are concerned, but to some technical extent also in OS installation. However, database systems are still engineered mostly monolithically for the largest possible feature set. We advocate that the idea of supporting a limited number of different "models" should already be a major consideration in the design and engineering of data management software.

4.2 Prerequisites of Success

Need for “Universal Glue”: The problem of composing different data managers into value-added application services is, of course, much more difficult than for standard consumer electronics. As noted earlier, we have to make sure that the complexity that we aim to reduce by simplifying the components does not reappear at the application layer and may become even more monstrous than ever. Simple interfaces with limited functionality and standardized cross-talk protocols are a fundamental prerequisite for composability and manageability of composite systems. Thus, higher-level application servers do need some standardized form of middleware such as OLE-DB or EJB to be able to talk to each underlying data server in a uniform manner. Such “universal glue” is available today. In particular, it is now a standard exercise to coordinate distributed transactions across arbitrary sets of heterogeneous servers by means of standardized 2PC protocols. So one important historical reason for putting all mission-critical data into a centralized, “universal” database for consistency preservation has become obsolete. Even when the classical ACID properties are inadequate for the application at hand, workflow technology is about to become mature and can reliably orchestrate activities on arbitrary servers within long-lived business processes. This is not to say that each and every technical aspect of how to combine arbitrary services into transactional federations is perfectly

understood (e.g., see [1] for open issues), but most of the issues are settled and for the remaining ones research solutions are at least within reach.

Apply Occam's Razor: Following Occam's philosophy, we should be extremely careful, even purists, in selecting which features a data manager should support and which internal mechanisms it needs to have to this end, aiming to minimize the complexity of both *interfaces and internals*. Often certain involved features can be implemented on top of a RISC data manager with a moderate loss of performance and a minor increase in programming efforts. So the gain from using a database system would be slightly reduced, but, at the same time, the pain of having to manage a feature-overloaded, complex system could be drastically alleviated. So the win is in improving the gain/pain ratio of database technology. For example, one could argue that null values with all their ramifications should be implemented by the application rather than the underlying data manager: the application understands its specific semantics of missing or undefined values much better, and this would also eliminate a multitude of null-related, often tricky rewrite rules in the data manager's query optimizer.

Likewise, we should avoid an unnecessarily broad repertoire of implementation mechanisms within a single-table, SPJ, or SPJ+Aggregation processor. Often certain mechanisms improve performance by only moderate factors and only in special cases, but significantly add to the complexity of system tuning. For example, hash indexes and hash-based query processing (including the popular hash joins) are probably never better than a factor of two compared to nested-loop variants (including those with index support etc.) or sort-merge-based query processing [14]. Similar arguments could probably be made about pipelined query execution (especially on SMPs with very large memory where the performance impact of pipelining is noise compared to that of data parallelism and may even hamper processor-cache effectively), fancy notions of join indexes etc..

Need for a Self-Tuning Framework: A major incentive for moving towards RISC style data managers is to enable auto-tuning of database components. As explained earlier, tuning must consider the relationship between workload characteristics, knob settings, and the resulting performance in a quantitative manner. Therefore, it is not surprising that the most promising and, to some extent, successful approaches in the past decade have been based on mathematical models and/or feedback control methods (e.g., to dynamically adapt memory partitioning or multiprogramming levels to an evolving, multi-class workload). Unfortunately, these models work only in a limited context, i.e., when focusing on a particular knob (or a small set of inter-related knobs). Attempting to cover the full spectrum of tuning issues with a single, comprehensive model is bound to fail because of the lack of sufficiently accurate mathematical models or the intractability of advanced models. This is why limiting ourselves to using only RISC data managers is so important: we no longer need to aim for the most comprehensive, elusive performance model, and there is hope that we can get a handle on how to auto-tune an individual data server component. It is much easier to tune a system with a less diverse workload and less dynamic resource sharing among different data and workload classes. Of course, the global tuning problem is now pushed one level above: how do we tune the interplay of several RISC data managers? Fortunately, a hierarchical approach to system tuning appears to be more in reach than trying to solve the entire complex problem in one shot. The main steps of such a hierarchical self-tuning framework are: 1) identifying the need for tuning, 2) identifying the bottleneck, 3) analyzing the bottleneck, 4) estimating the performance impact of possible tuning options, and 5) adjusting the most cost-effective tuning knob.

The hierarchical nature of such a self-tuning procedure is perfectly in line with good practice for manual, or we should better say intellectual, tuning (e.g., [16, 20, 21]). In particular, our approach also adopts a "think globally, fix locally" regime. Further note that mathematical models have been and remain to be key assets also in the practical system tuning community (e.g., [11]). The key to making the mathematics sufficiently simple and thus practical lies in the reduced complexity of the component systems and their interfaces and interplay. We believe that there is a virtue in engineering system components such that their real behavior can be better captured by existing mathematical modeling techniques, even if this may lead to some (tolerable) loss of high-end features and efficiency. The true gain lies in the better predictability and thus tunability and manageability of systems.

5. Towards a Research Agenda

5.1 Evaluation of Success

How should we evaluate the viability and success or failure of the advocated architecture? Actually evaluating an architectural framework would entail designing, building, and operating nontrivial IT systems, which obviously is way beyond the scope of a paper (especially a semi-technical vision paper like this one). So we merely give a rough sketch of how we as a research community may proceed along these lines.

The best measure of success of a RISC-style database system architecture would be to demonstrate the usefulness of the components in a variety of data management scenarios. To start with, we should be able to develop data management components that work well as scalable traditional OLTP systems as well as the basis for OLAP-style data management services. Such systems are likely to use the SPJ query processor and the SPJ+Aggregation query processor, respectively.

Incidentally, recall that the first generation of OLAP services extensively used "multi-statement" SQL, i.e., they fed only simple queries (SPJ with aggregation) to backend servers to ensure that performance characteristics are predictable since database optimizers have been known to behave erratically for complex SQL queries (which should not really be a surprise given that a typical query optimizer is a large bag of tricks).

Another interesting instance of services is metadata management. Such a service is distinguished from traditional OLTP and OLAP in requiring more elaborate conceptual data models, but with relatively simple needs for querying and scalability (for large data volumes). Yet another popular class of services is management of e-mail data. Mail servers require fast insertion of mail messages and "mostly fast" access when queried by attributes. A key characteristic of mail data is that it is sparse, i.e., not all attributes are present and records are of widely variable length. It would be intriguing to consider an SPJ query engine and a separate indexing engine as a mail server's base components. In contrast to traditional relational system designs, however, there should be no limiting necessity of a schema in this setting. Other data services of interest, whose architecture would be worthwhile to re-examine with this paper's philosophy of RISC building blocks in mind, are marketplace data servers (e.g., in E-Bay) or large-scale multimedia information servers for news-on-demand etc.

5.2 Research Opportunities

We wish to encourage researchers to make the system architecture of database technology and the simplification of component interfaces as well as internals (again) a top-priority item on their agenda. Although most of the problems with today's architecture that we have identified refer to industrial products, we believe that the impetus for a new departure must come from academia as product architects and developers are way too busy in their time-to-market issues. To be fair, the database system industry has a lot of stake in maintaining existing products and market shares, and has too little leeway for radical departures. For academic researchers we see ample opportunities of system-oriented exploration that can hopefully drive the community towards a better overall architecture. It is crucial, however, already for component prototyping and systematic experimentation that this research avenue heads for simpler, smaller-scale, RISC-style building blocks.

Along these lines, we propose making major efforts towards the following research (and, to some extent, sociological) challenges:

- Make viable an open, worldwide testbed for RISC-style data-management components to which even small research teams can contribute.
- Work out lean APIs for each of the most important RISC-style components following our discussion of Section 4.1 on different kinds of query processors and storage managers.
- Encourage a worldwide competition for the "best" instantiation of each of these building blocks, for example, with regard to certain standard benchmarks or common test suites for data mining and Web applications. To make this a real challenge and avoid compromising our goal of simplified and auto-tuned components, the rules should limit the code size of each component, limit its footprint, and disallow any kinds of tuning knobs other than what the component does internally based on its own self-assessment.
- To ensure that individual components are not tailored to other components by the same team or even tailored to specific benchmarks, all components that are registered with the worldwide testbed must be able to correctly cooperate, through their official APIs, with all other components from all other teams.
- Identify more precisely the "universal glue" for the above kind of open test bed. Obviously this is already required for setting up the test bed. A bootstrap approach could be that one team provides an initial instantiation of the necessary middleware services and the most important components as a basis for other teams to contribute and plug in individual components or gradually replace some of the "glue" services (e.g., a two-phase commit protocol).

Once the envisioned test bed is operational for database-system components, its scope could and should be broadened to encompass different kinds of data managers for IT solutions in the large (e.g., a media server, a mail server each again built from several RISC-style components). So this worldwide test bed should be extensible beyond the narrow boundaries of what is commonly perceived as the "database community".

6. Concluding Remarks

Universal database systems grew out of our belief that database is the center of the universe and therefore the framework for integration. This is far from true in today's world. Once we are liberated and can accept the fact that the database is one, certainly very important, component in the IT world, programmability and integration of database components with applications become a priority. In such a world, we need to build RISC-style data management components that have predictable performance and can be auto-tuned.

When comparing our field to other areas of engineering that are building extremely complex artifacts such as aircrafts, high-speed trains, or space shuttles, we realize that such an architectural simplification is overdue and critical for the future success of database technology. Few, if any, understand the functions of a modern aircraft (e.g. Boeing 747) completely, but in contrast to the

situation with database systems, there is excellent understanding of local components and a good understanding of the interaction across components.

The test for the advocated RISC-style database system architecture will be if it can be used broadly in many more contexts compared to today's database systems. In this paper, we have hinted at some applications that can drive the design of such RISC data management components. These components can be "glued" together using narrow APIs to form powerful data services that have the hope of being effectively manageable.

The bottom line of these challenges is to foster improving the "gain/pain ratio" of database technology with regard to modern cyberspace applications. The key to this goal is to tolerate a moderate degradation of "gain", for example, by tolerating certain overhead for more interface-crossing across components, and reduce the "pain" level by orders of magnitude by ensuring predictable performance and eliminating the need for manual tuning.

References

[1] G. Alonso, C. Hagen, H.-J. Schek, M. Tresch: Distributed Processing Over Stand-alone Systems and Applications, 23rd International Conference on Very Large Data Bases, Athens, Greece, 1997.

[2] P. Bernstein et al. : The Asilomar Report on Database Research, ACM SIGMOD Record Vol.27 No.4, Decemer 1998.

[3] D.S. Batory, T.Y. Leung, T.E. Wise: Implementation Concepts for an Extensible Data Model and Data Language, ACM Transactions on Database Systems Vol.13 No.3, 1988.

[44] K.P. Brown, M. Mehta, M.J. Carey, M. Livny: Towards Automated Performance Tuning for Complex Workloads, 20th International Conference on Very Large Data Bases, Santiago, Chile, 1994.

[5] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, E.J. Shekita: The Architecture of the EXODUS Extensible DBMS, in: K.R. Dittrich, U. Dayal (Eds.), International Workshop on Object-Oriented Database Systems, Pacific Grove, 1986.

[6] M.J. Carey, D.J. DeWitt: Of Objects and Databases: A Decade of Turmoil, 22nd International Conference on Very Large Data Bases, Bombay, India, 1996.

[7] S. Chaudhuri, V. Narasayya : An Efficient, Cost-driven Index Tuning Wizard for Microsoft SQL Server, 23rd International Conference on Very Large Data Bases, Athens, Greece, 1997.

[8] S. Agrawal, S, Chaudhuri, V. Narasayya: Automated Selection of Materialized Views and Indexes, 26th International Conference on Very Large Data Bases, Cairo, Egypt (this proceedings).

[9] D. Chatziantoniou, K.A. Ross: Querying Multiple Features of Groups in Relational Databases, 22nd International Conference on Very Large Data Bases, Bombay, India, 1996.

[10] S. Chaudhuri (Editor): IEEE CS Data Engineering Bulletin, Special Issue on Self-Tuning Databases and Application Tuning, Vol.22 No.2, June 1999.

[11] Computer Measurement Group, Inc., http://www.11.org

[12] A. Geppert, K.R. Dittrich: Bundling: Towards a New Construction Paradigm for Persistent Systems, Networking and Information Systems Journal Vol.1 No.1, 1998.

[13] G. Graefe, D.J. DeWitt: The EXODUS Optimizer Generator, ACM SIGMOD International Conference on Management of Data, San Francisco, 1987.

[14] G. Graefe: The Value of Merge-Join and Hash-Join in SQL Server. 25th International Conference on Very Large Data Bases, Edinburgh, UK, 1999.

[15] L. Haas et al.: Starburst Midflight: As the Dust Clears, IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1, 1990.

[16] D.A. Menasce, V.A.F. Almeida: Capacity Planning for Web Performance – Metrics, Models, and Methods, Prentice Hall, 1998.

[17] R. Munz: Usage Scenarios of DBMS, Keynote, 25th International Conference on Very Large Data Bases, Edinburgh, UK, 1999, http://www.dcs.napier.ac.uk/~vldb99/Industrial SpeakerSlides/SAPVLDB.pdf

[18] US President's Information Technology Advisory Committee Interim Report to the President, August 1998, http://www.ccic.gov/ac/interim

[19] H.-J. Schek, H.-B. Paul, M.H. Scholl, G. Weikum: The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1, 1990.

[20] T. Schneider, SAP R/3 Performance Optimization: The Official SAP Guide, Sybex, 1999.

[21] D.E. Shasha: Database Tuning: A Principled Approach, Prentice Hall, 1992.

[22] A. Silberschatz, S. Zdonik, et al.: Strategic Directions in Database Systems - Breaking Out of the Box, ACM Computing Surveys Vol.28 No.4, December 1996.

[23] M. Stonebraker, L.A. Rowe, M. Hirohama: The Implementation of Postgres, IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1, 1990.

[24] G. Weikum, C. Hasse, A. Moenkeberg, P. Zabback: The COMFORT Automatic Tuning Project, Information Systems Vol.19 No.5, 1994.