SET-STORE DATA ACCESS ARCHITECTURES

For High Performance Informationally Dense I/O Transfers

D L Childs
Integrated Information Systems
Ann Arbor, Michigan
http://xsp.xegesis.org/
iis@umich.edu

ABSTRACT

For high performance analytic processing of vast amounts of data buried in secondary storage, traditional performance strategies generally advocate minimizing system I/O. This paper advocates the converse supported by the use of set-store architectures. Traditional row-store and column-store architectures rely on mechanical data models (based on an imposed physical representation of data) for accessing and manipulating system data. Set-store architectures rely on a mathematical data model (based solely on the intrinsic mathematical identity of data) to control the representation, organization, manipulation and access of data for high performance informationally dense parallel I/O transfers.

1. INTRODUCTION

This paper is concerned with the architecture of analytic processing applications that require access to very large amounts of remotely stored data. Specifically, the paper's focus is on that portion of the system architecture that stores and accesses data required for application execution.

Three major components are common to every databased system architecture: the data repository, the application, and the data access mechanism connecting them. How data is represented, manipulated and exchanged between these components determines the functionality and performance of a system.

Critical to system performance is the ability to serve the functional requirements of the application in harmony with the data access mechanism and the storage organization of the repository. This is the challenge when developing a system and like any architectural development, it helps to have a *blueprint*. The blueprint is a system data model.

1.1 System Data Models

Data models provide a conceptual understanding of how data is to be represented and manipulated. Application processing models focus on data representations and data manipulations that serve functionality requirements. Data repository models focus on data representations suited for storing large quantities of data and operations for quickly locating and transferring relevant data. Unfortunately, the data representations and operations of interest for one are not usually compatible with those of the other. The role of a system data model is to provide a synergistic integration of an application data model with a repository data model.

Data models come in two flavors: mechanical (concrete) or mathematical (abstract). Mechanical data models rely on how the data *looks* by defining data representations and manipulations in terms of an imposed physical structuring of data items. Mathematical data models rely on what the data *is* by defining data representations and manipulations in terms of the intrinsic relationships among data items. Mechanical data models are dependent on knowledge of the physical representation of stored data which severely complicates system design. Mathematical data models are only dependent on knowledge of data relationships and thus allow the greatest flexibility in system design. Since there is no established mathematical model for data repositories, in practice system data models are hybrids of the two.

XSP Technology Research & Development Series [SSDAA: 05/13/10]
Copyright 2010 Integrated Information Systems.

1.2 Sets & Set Operations

This paper introduces a mathematical system data model based on the abstract primitive *set*. Unlike mechanical primitives that have a concrete visual association, abstract primitives do not. This is an acknowledged drawback of all mathematical data models. On the plus side mathematically modeled systems no longer need burden the application with knowledge of the repository data representations. System developers are now free to treat system performance options independent of application functionality. Set-store based system architectures can be shown to be a high performance alternative to traditional data access architectures.

Using set operations at the data repository level offers the performance potential for leveraging I/O throughput. Current systems focus either on leveraging processor speed or RAM size. Neither takes advantage of the potential offered by utilizing the available number of I/O paths and saturating I/O transfers with informationally dense data.

2. SYSTEM ARCHITECTURES

All databased system architectures support a user-friendly mechanism for extracting data relationships from a machine-friendly data repository. The representation of data in the repository is key to how well any specific system can access relevant data in the most timely fashion.

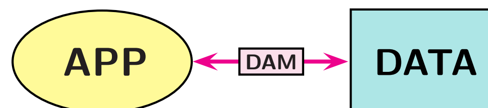

Figure 1: DAM: Data Access Mechanism

Though all databased system architectures are structurally the same, specific application requirements may vary greatly. Common to all is a data access mechanism, DAM, that recruits relevant data from storage and presents it in a usable

form to the soliciting application. Figure 1. depicts the relationship between the critical components of a system architecture.

Overall system performance is dominated by application data access requirements. Data access is potentially restricted by the inability of the repository to present timely relevant data to the application as required. Thus, overall system performance often depends on how well data exchange is mitigated by a data access mechanism.

2.1 DAM Performance

Many factors influence the performance of a data access mechanism, DAM, in achieving its purpose: to get the *right* data to the *right* place, in the *right* form at the *right* time. Each of these *rights* has a complex contribution to the total DAM performance. The best performing DAM is one that is not needed at all. Though this can be achieved with certain system architectures, it is a limiting architectural solution.

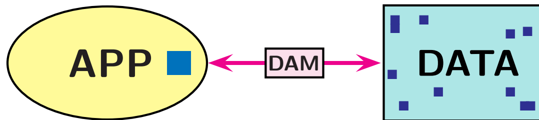

Figure 2: Mapping repository data to application data.

There is always a mismatch between data representations needed for application processing and data representations best suited for repository access. Typically, data required by an application is not generally in the right form at a single location in a repository, Figure 2. Even if all the right data is in the right form at the right time, it still has to be moved to the right place. The limiting performance factor with disk repositories is the time it takes to move data between disks and applications.

2.2 DTR: Data Transfer Rate

For any given amount of data that has to be transfered, the minimum elapsed time is determined by the best data transfer rate, DTR, available. Therefore, one criteria for measuring a system's performance is by how close a specific implementation approaches a platform's *optimal* DTR using a single I/O port.

A more meaningful measure of system performance is the total elapsed time to complete the transfer of a given amount of data in parallel with multiple I/O ports. With today's multi-core, multi-port platforms DAM architectures that can dynamically repartition repository data are best suited for providing near optimal DAM performance. It will be shown later that for many applications requiring access to vast amounts of repository data (possibly highly distributed) if a desired total elapsed time, TET, can be determined then a hardware platform can likely be chosen to support it.

2.2.1 Leveraging I/O Throughput

Today a single I/O port can easily sustain DTRs of 500 MB/sec. Two parallel ports can support 1,000 MB/sec. With only eighteen I/O ports a terabyte of data can be transfered to an application in one minute. With DTRs being able to support data transfers of this magnitude in this time frame, why are commercial systems taking hours[1] to process smaller amounts of data using larger platforms? Part of the answer is that no current DAM architecture leverages I/O throughput.

[1]see TPC-H benchmarks

2.3 DAM Buffer Size

An ideal high performance DAM architecture would transfer just the data required by the application using as many parallel ports as needed. Unfortunately, the DTR is not the only performance limiting consideration. Since applications can not generally accept all required data in one transfer, transfers are broken up into a sequence of I/O buffer blocks. The number, size, and location of these buffers greatly impact total elapsed times.

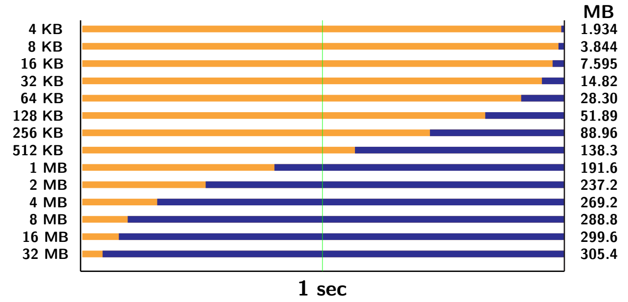

Figure 3: Buffer size impact on randomly accessed data.

Since there is search overhead for locating each buffer, minimizing the number of required buffers gives the best access times. When large amounts of data are required, the larger the buffer size the better the performance. Figure 3. compares the impact of buffer size choice on the amount of data that can be accessed in one second, given a disk environment with a sustainable DTR of 320 MB/sec.

In Figure 3. the cost (in elapsed time) for locating buffers to be transfered, is represented in orange. The actual data transfered is represented in blue. For any I/O transfer buffer under 1 MB, the system spends more time looking for data than in transferring data. A buffer size of less than 1 MB is clearly sub-optimal for accessing large volumes of data. Conversely, using a 32 MB allows a system to access a terabyte of data in less than an hour, using just one I/O port. Using forty-eight[2] I/O ports a terabyte of data can be accessed by an application in a little over one minute.

In summary, the architectural function of a DAM is to provide an application the data it wants, when it wants it, where it wants it, and with the best possible performance. DAM performance, in turn, is dependent on the repository support which is a function of system DTR and strategic use of I/O buffers. Though there is always an expected mismatch between data representations best suited for application processing and data representations best suited for repository access, some common ground has to be shared between the two. All of which is dictated by the application requirements.

3. APPLICATION REQUIREMENTS

DAM performance is based on locating and transferring the *right* data in the *right* form to the *right* place. For every design, the specification of what constitutes the *right* data, the *right* form, and the *right* place is established by the needs of the application. Therefore, the performance options of the DAM component of an architecture are dictated by the demands of the application processing requirements.

[2]a number compatible with TPC-H 1TB participants.

3.1 Application Requirement Minimum

What ever an application objective might be, there is always a requirement for processing a specific collection of data. Accessing this data, and only this data, from a data repository is an architectural and implementation challenge. Accessing more data than needed does not affect the result, but could severely affect performance. Not accessing all the required data is not an option

For exposition the term application required minimum, ARM, is defined to be the minimum collection of data that needs to be examined by a specific application. Since any given ARM represents the minimum amount of repository data required for transfer, the performance goal of every system DAM is to transfer the smallest amount of data, that includes the ARM, in the shortest possible time.

3.2 Buffer Management

As long as an application requires data from a repository the dominant performance constraint will be the number, size and content of the I/O buffers used. An optimal ARM supported implementation would only transfer application required data, and in the shortest possible time.

In an analytic processing system a typical ARM would be many megabytes oriented rather than several bytes oriented. Figure 3 indicates that a few large I/O buffers transfer more data per second then many small I/O buffers. Optimal buffer management would dictate the larger the I/O buffers used, the better the performance. Figure 3. also assumes that all data transfered is required data. In practice this is an unattainable ideal.

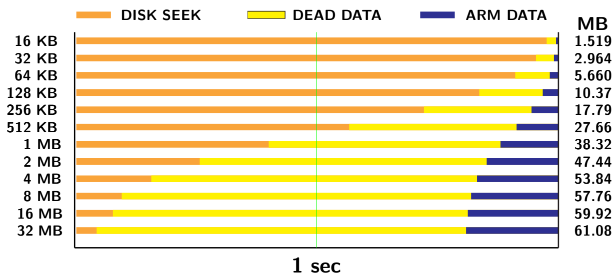

Figure 4 20% Relevant Data Transfers.

Figure 4. reflects buffer usage where only 20% of data transfered is required data. A single data repository must support a barrage of varying ARM requests. The less control offered by buffer management, the lower the relevant data percentage is likely to be. The choice and deployment of buffer management strategies is the most important task of a system DAM. The representation and organization of repository data determines the options available for buffer management.

3.3 Repository Data Organization

The ideal repository organization would be to provide application requests with buffer I/O transfers containing only the data required by the application. Even coming close to this ideal has traditionally been accepted as an extremely difficult and challenging architectural task. Part of the difficulty is intrinsic to the nature of the task itself and part is due to the tools and approach traditionally used to seek and employ an architectural solution.

The data organization task is intrinsically difficult since repository organization is dependent on the application data processing requirements and the application data processing requirements are dependent on the application objectives. Thus, the organization of repository data has to anticipate ARM needs or be able to adapt quickly to application directives. If application objectives are known sufficiently in advance the task becomes more manageable, but with ad hoc queries on fresh data, the task of optimizing repository data for best performance becomes nearly intractable.

Since the best organization of repository data is ultimately dependent on user specified application intent, the intent has to somehow be communicated to a *repository optimizer* prior to an application's request for required data. To optimize I/O response the repository optimizer must also know the form of the data needed by the application. Since this is a processing parameter and not a specification requirement, the repository optimizer needs clues from the application as to the data representation required. Since data content is provided by a user and data representation for processing is provided by the application, some means is needed to capture both data *content* and *representation* information to use in organizing repository data for optimum I/O access.

Since any specific repository is likely to service many simultaneous applications and since the ARMs are likely to be varied (even incompatible), the system specification is likely to be quite demanding. What ever set of tools chosen, they need to be able to express a variety of data representations and manipulations of these representations.

Since application data representations are intended for processing purposes and since repository representations are intended for mass storage and rapid retrieval support, it is unlikely that representations shared between them would be an optimal choice for either. Given data representations best individually suited for data processing and for data preservation, the challenge for system data models is in describing how to *share* data between them.

4. SYSTEM DATA MODELS

What may be the greatest deficiency in the development of software systems is the lack of *any* formal system data model for specifying application intent, translating that intent into executable routines, and supporting preserved data for access by those routines. This is not a condemnation of the efforts devoted toward remedying this modeling deficiency, but rather a recognition of the difficulty of the challenge.

To fully appreciate this difficulty it is necessary to agree on exactly what constitutes a model, what is to be modeled, and what is lacking in current system data models.

4.1 Data Models

A *model* is a formal abstract system of objects with rules for the manipulation of objects. In a *data model* the objects are representations of data relationships and the rules dictate how these data representations are manipulated.

4.2 System Components

For any system model to be complete all the components of the system must be contained in the model along with the modeling of the interactions between system components.

A typical system is comprised of three major interacting components, or subsystems: 1) a subsystem of data objects and their rules for specifying an application objective; 2) a subsystem of data objects and their rules for electronically realizing the application objective; and 3) a subsystem of

data objects and rules for providing rapid access to preserved data representations.

4.2.1 Data Sharing

The difficulty with modeling any system involving users, processors, and secondary storage environments is being able to pick the data representation that best accommodates the needs of each environment. This, in turn, is complicated by the system need to *share* data between the environments.

The most demanding feature of system data models is the ability to share, between system subsystems, information about data content without exposing information about data representations. The term *Data Independence*[3] between two data modeling environments asserts that information about data content and behavior is shared between them, but the structural properties[4] of the data representations are not. The term *structural isolation* may be a more accurate modeling term than data independence.

The system modeling challenge is providing best overall system performance while enforcing functional integrity. All three subsystems are working conjunctively on the same data relationships, but each requires a different physical representation of the data relationships to support best performance. The difficulty arises when models are chosen that use data structure sharing in order to share knowledge of data relationships.

4.3 System Data Independence

Figure 5. displays the evolution of data independence in terms of where systems were, where they are, and where the could be. USR represents application specifications, EXC application processing, and the modeling of repository data organization by STO.

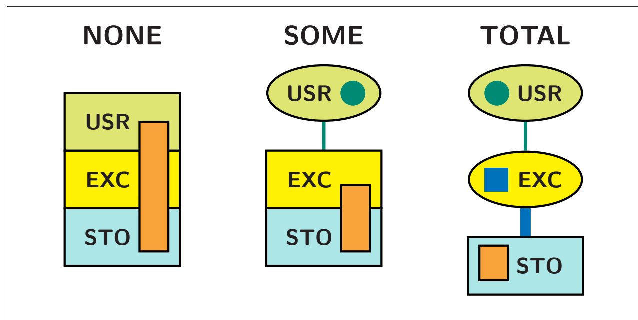

Figure 5: Degrees of Data Independence

The Hallmark of pre-Codd WYSIWYG architectures was sharing knowledge of the physical representation and the organization of data across all three subsystem levels.

Most current architectures still force knowledge of the physical representation and organization of data to be shared by both the execution and the storage environments, but do provide some[5] degree of data independence to users.

System data independence requires the representation and organization of data at each of the three levels be unknown to each of the other two, while data content is available to all three.

4.4 Data Content

Data independence means only that knowledge of data representations is not shared between subsystems. It does not mean that knowledge of data content[6] is not shared. Subsystems must share and preserve data content for any system to work. With shared data representations data content is also shared. The research conundrum is how to share data content without sharing data representations.

The evolution from past to present systems was made possible by discovering how to capture the *mathematical identity*[7] of data content as *relations* and the development of operations to manipulate these relations [Co70].

Present systems provide a degree of data independence between user specification and application execution, but application execution and repository data environments are still shackled by data representation sharing. Though data independence is lacking at the I/O interface in present systems, it need not be the case in future systems.

An evolution from current, partially data independent systems, to future, totally data independent systems, requires a similar discovery for capturing the mathematical identity of the repository data. Since the data content of the repository data is itself a data representation containing application data representation and content, any adequate mathematical data model will, by necessity, have to formally model both data content and data structures.

4.5 System Bridge Models

If two environments are truly data independent, in that they share no knowledge of each other's data representations or data organization, then some data access mechanism is needed to bridge them that supports data content exchange.

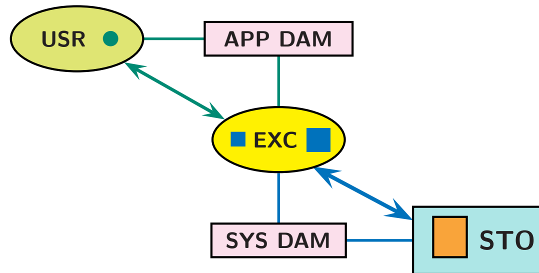

Figure 6: Structure Morphing Bridge Mechanisms

Two structure morphing bridge mechanisms are needed to complete a system data model. An application data morphing mechanism is needed to transform data structures between programming and execution environments, and a system data morphing mechanism is needed to transform data structures between execution and storage environments.

4.6 RDM System Data Models

According to Figure 6, five separate data models[8] are required to completely define an RDM system data model. The data representations for the three principle subsystems are well known: Tables, arrays, and files. What is not well

[3] introduced by Codd as application ignorance of system data representations [Co70] and extended here to mean data representation ignorance between any two environments.

[4] "order dependence, indexing dependence, and access path dependence." [Co70]

[5] some current implementations violate data independence with application knowledge of *index structures*.

[6] relationships between data items

[7] a set-theoretically defined object [Ch07]

[8] separate systems of objects and rules

known is how to best morph one structure to another to achieve best performance while preserving data content.

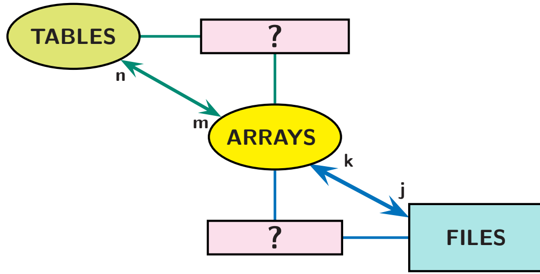

Figure 7: Mapping Relational System Data Representations

The transmutation from Tables to arrays and from arrays to files is a dominant challenge for performance concerned system architects. The challenge is in adopting the best data representation for each system component as needed during execution. This is a performance challenge, not a *verifiably correct result* challenge.

4.7 RDM System Performance

The performance challenge for the system architect is in choosing the best data representation for each of three specific purposes:

- Programming Productivity
- Data Processing
- Data Preservation

Each of these benefits from a specific data representation:

- Programming Productivity: TABLES
- Data Processing: RC-ARRAYS
- Data Preservation: FILE STRUCTURES

When there are n possible Tables for best productivity, m possible RC-arrays for best execution, k possible RC-arrays for best execution support, and j possible representations for best data access, the control of structure transmutations for overall optimal performance becomes quite challenging.

5. ARRAYS & TABLES

Arrays are the quintessential representations for data relationships. Arrays are simply defined as items structured in rows and columns. An n-row by k-column array is easy for users to visualize, it is a natural way to capture data relationships having k distinct properties, and it is easily represented electronically by sequences of bytes.

In early systems the programmer, processor, and storage were inextricably linked together by a single concrete array representation of data relationships. The programmer was viewed as a *navigator* [9] equipped with seven tactics for locating and accessing relevant rows of an array.

5.1 Arrays

Arrays are so conceptually simple that requiring them to have a mathematical identity seems, not only superfluous, but unnecessary, inconvenient, and unproductive. All of which is certainly true at the conceptual level.

The contribution of a mathematical identity is to ensure valid mapping of both content and structure of a conceptual data representation to an electronic data representation. Any such mapping requires that both the conceptual and the electronic representations be formally well defined by means of a mathematical identity.

5.1.1 Arrays as Sets

Abstract arrays already have a familiar mathematical identity as an ordered sequence of *rows*. Rows in turn are commonly represented by tuples. Any abstract array can easily be captured by an n-tuple of k-tuples. Where n is the number of rows in the array and k is the number of columns.

$$\mathbf{A}_1 = \begin{array}{|c|c|c|}\hline a & b & c \\ \hline x & y & z \\ \hline \end{array} \quad = \quad < < a, b, c >, < x, y, z > >$$

$\mathbf{A}_1$ above visually and set-theoretically expresses both the content and structure of a 2-row 3-column array. When a data item (having both content and structure) is to be embedded into a structured environment the structure imposed on the embedded data item has to be given a formal representation. A formal representation of $\mathbf{A}_1$ embedded in an addressable machine environment has to preserve the original data content and data structure of the array and include the new data item structure in the address space.

All that is really necessary is the addition of decipherable *location notation* that formally reflects the embedded data item structuring. Following are two of many candidates for embedding $\mathbf{A}_1$ in a well-ordered environment:

$$\mathcal{E}_1(\mathbf{A}_1) \ = \ \{< Ad_1, < a, b, c >>, \ < Ad_2, < x, y, z >>\}$$

$$\mathcal{E}_2(\mathbf{A}_1) \ = \ \{< a, b, c >^{Ad_1}, \ < x, y, z >^{Ad_2} \ \}$$

Both notations pair an address with a string of bytes. The value of the addresses may serve to preserve the row order of $\mathbf{A}_1$ as an array, or to impose a row access order of interest, or be considered irrelevant as in a Table. How this association is actually implemented is not relevant to the logical description of the model, but is most likely to dramatically impact system performance.

5.1.2 Content Equivalent Arrays

Though arrays have a conceptually friendly abstract representation that can easily map to a concrete machine representation, arrays do not uniquely represent data relationships, due to the implicitly imposed orderings on both rows and columns.

Given two items, in some reality of interest, each having three distinguishing properties, there are 12 structurally distinct arrays expressing the same data relationships.

$$\mathbf{A}_1 = \begin{array}{|c|c|c|}\hline a & b & c \\ \hline x & y & z \\ \hline \end{array} \quad \mathbf{A}_2 = \begin{array}{|c|c|c|}\hline x & y & z \\ \hline a & b & c \\ \hline \end{array} \quad \mathbf{A}_3 = \begin{array}{|c|c|c|}\hline a & c & b \\ \hline x & z & y \\ \hline \end{array} \quad \mathbf{A}_4 = \begin{array}{|c|c|c|}\hline x & z & y \\ \hline a & c & b \\ \hline \end{array}$$

$$\mathbf{A}_5 = \begin{array}{|c|c|c|}\hline b & a & c \\ \hline y & x & z \\ \hline \end{array} \quad \mathbf{A}_6 = \begin{array}{|c|c|c|}\hline y & x & z \\ \hline b & a & c \\ \hline \end{array} \quad \mathbf{A}_7 = \begin{array}{|c|c|c|}\hline b & c & a \\ \hline y & z & x \\ \hline \end{array} \quad \mathbf{A}_8 = \begin{array}{|c|c|c|}\hline y & z & x \\ \hline b & c & a \\ \hline \end{array}$$

$$\mathbf{A}_9 = \begin{array}{|c|c|c|}\hline c & a & b \\ \hline z & x & y \\ \hline \end{array} \quad \mathbf{A}_{10} = \begin{array}{|c|c|c|}\hline z & x & y \\ \hline c & a & b \\ \hline \end{array} \quad \mathbf{A}_{11} = \begin{array}{|c|c|c|}\hline c & b & a \\ \hline z & y & x \\ \hline \end{array} \quad \mathbf{A}_{12} = \begin{array}{|c|c|c|}\hline z & y & x \\ \hline c & b & a \\ \hline \end{array}$$

Figure 8: 12 Content Equivalent Arrays.

For any given array with n-rows and k-columns there are $n! \times k!$ arrays expressing the same item relationships. Since

[9] an under-appreciated non-mathematical approach to the problem of accessing repository data [Ba73]

the ordering and structure of the representations of data relationships in no way affects the data relationships, the programmer should be able to reference the data content directly and not be burdened by having to chose the best imposed structuring.

If there were a definition of an array that formally acknowledged the distinction between data content and data structure, then an operation $\mathcal{DC}(\mathbf{A})$ could be defined that expressed just the data content of array, $\mathbf{A}$. Then for all $\mathbf{A}_i$, $\mathbf{A}_j$ in Figure 8, $\mathcal{DC}(\mathbf{A}_i) = \mathcal{DC}(\mathbf{A}_j)$. This is exactly what motivated relational Tables.

5.2 Tables

RDM-Tables[10] are abstract data structures intended to provide a structure independent representation for modeling complex data relationships. The structure of RDM-Tables resembles that of an array with rows and columns properly interpreted to reflect data relationships as *relations*[11]. The following twelve RDM-Tables all represent the same relation.

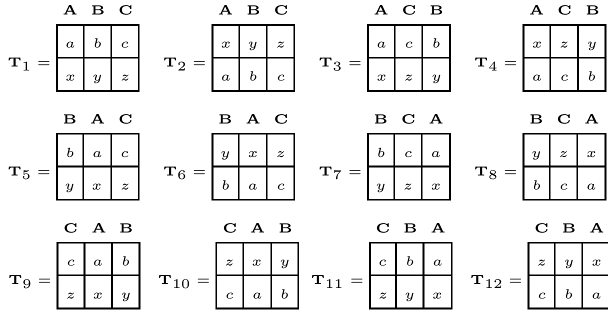

Figure 9: Content Equivalent RDM-Tables.

Since RDM-Tables are abstract structures, none of the above twelve representations has physical substance. Yet any one of the twelve can be mapped to an array with an *understanding* that a specific ordering of array columns reflects a specific ordering of RDM domains. To replace an implied understanding with a formal model requires a mathematical identity for all $\mathbf{T}_i$, $\mathbf{T}_j$ in Figure 9 such that $\mathcal{DC}(\mathbf{T}_i) = \mathcal{DC}(\mathbf{T}_j)$. This requires that all Tables have a mathematical identity.

5.2.1 Tables as Sets

For any formal mathematical modeling between TABLES and ARRAYS (Figure 7) to exist, both Tables and arrays need to have mathematical identities. A complete formal description of Tables as sets can be found in [Ch07], though only a few notational equivalences are now required:

$$< x_1, x_2, .., x_n > \equiv \{x_1^1, x_2^2, .., x_n^n\}$$

$$< x_1, x_2, .., x_n >_{<S_1, S_2, .., S_n>} \equiv \{x_1^{S_1}, x_2^{S_2}, .., x_n^{S_n}\}$$

The mathematical identities for three of the twelve RDM-Tables can now be formally expressed by:

$$\mathbf{T}_1 = \left\{ < a, b, c >^1_{<\mathbf{A},\mathbf{B},\mathbf{C}>},\ < x, y, z >^2_{<\mathbf{A},\mathbf{B},\mathbf{C}>} \right\}$$

$$\mathbf{T}_6 = \left\{ < y, x, z >^1_{<\mathbf{B},\mathbf{A},\mathbf{C}>},\ < b, a, c >^2_{<\mathbf{B},\mathbf{A},\mathbf{C}>} \right\}$$

$$\mathbf{T}_{12} = \left\{ < z, y, x >^1_{<\mathbf{C},\mathbf{B},\mathbf{A}>},\ < c, b, a >^2_{<\mathbf{C},\mathbf{B},\mathbf{A}>} \right\}$$

[10]Originally introduced by Codd as *array representations* of relations, for expository purposes only. [Co70]

[11]Codd used the term *relation* in its strict set-theoretic sense as a non-empty subset of a Cartesian product. [Co70]

The extracted data content of any one of the Tables is:

$$\mathcal{DC}(\mathbf{T}_i) = \left\{ \ \{a^{\mathbf{A}}, b^{\mathbf{B}}, c^{\mathbf{C}}\},\ \{x^{\mathbf{A}}, y^{\mathbf{B}}, z^{\mathbf{C}}\} \ \right\}.$$

The above operation satisfies the first requirement of a formal application data morphing (Figure 6) from TABLES to ARRAYS. It allows n structurally distinct Tables to map to 1 formal expression of Table data content.

5.3 Table-Arrays

The $\mathbf{A}_i$ arrays of Figure 8 are easily expressed with rows and columns, RC-arrays, in concrete terms of byte strings and in abstract terms as nested tuples. What RC-arrays do not capture is the RDM Domain relationship with column orderings.

Table-Arrays (or T-Arrays) are defined by ordered pairs expressing a RC-array and a column-Domain mapping tuple having the form: $\mathbf{TA}_i = < \mathbf{D}_i, \mathbf{A}_i >$. Where $\mathbf{A}_i$ is a RC-array of the form in Figure 8 and $\mathbf{D}_i$ is a tuple specifying desired domain orderings. For example:

$$\mathbf{TA}_1 = << \mathbf{A}, \mathbf{B}, \mathbf{C} >, < a, b, c >, < x, y, z >>$$

$$\mathbf{TA}_6 = << \mathbf{B}, \mathbf{A}, \mathbf{C} >, < y, x, z >, < b, a, c >>$$

$$\mathbf{TA}_{12} = << \mathbf{C}, \mathbf{B}, \mathbf{A} >, < z, y, x >, < c, b, a >>$$

For any unique data relationship to be captured by an RDM-Table there are $k!$ distinct[12] RDM-Table structural representations. For each RDM-Table there are $n!$ distinct[13] RC-arrays. Since just one of the RDM-Tables may be best suited for programmer productivity and since just one of the RC-arrays may be the best choice for program execution, the ideal system data model should allow bi-directional optimization strategies to connect the best of both worlds.

Since the data content remains the same in both, only the structure of the data relationship representation needs to be manipulated. Given that each environment is aware of its structural preference, all that is required of a cooperative mapping capability is a formal means for recognizing the common data content, and the individually desired data representation structures on each end of the mapping.

5.3.1 Mapping Tables to Table-Arrays

Since one end[14] of the mapping contains Tables, and since the other end[15] contains arrays, the first requirement for any content preserving mapping formalism is the ability to support the content preservation mapping of any chosen $\mathbf{T}_i$ of Figure 9 to and from any $\mathbf{TA}_j$ Table-array equivalent RC-array of Figure 8.

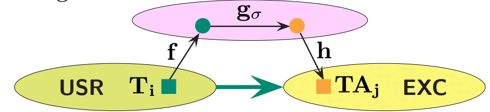

Figure 10: Content Preserving Table-Array Mappings

According to Figure 10, for any $\mathbf{T}_i$ and any $\mathbf{TA}_j$ there exists a $\mathbf{f}$, a $\mathbf{g}_\sigma$, and a $\mathbf{h}$ such that $\mathbf{h}(\mathbf{g}_\sigma(\mathbf{f}(\mathbf{T_i}))) = \mathbf{TA}_j$. Where:

- $\mathbf{f}(\mathbf{x})$ removes Table structuring of $\mathbf{x}$,
- $\mathbf{g}_\sigma(\mathbf{y})$ adds σ-Domain structuring to $\mathbf{y}$, and
- $\mathbf{h}(\mathbf{z})$ adds row-order structuring to $\mathbf{z}$.

[12]where k is the number of Domains

[13]where n is the number of rows

[14]formally called the *domain*

[15]formally called the *range*

For example, assume set-theoretic functions have been defined[16] for mapping Tables to Table-Arrays, then mapping $\mathbf{T_1}$ to $\mathbf{TA_6}$ might proceed as follows with $\sigma = <B, A, C>$:

$$\begin{aligned}
\mathbf{T}_1 &= \{ <a, b, c>^1_{<\mathbf{A},\mathbf{B},\mathbf{C}>},\ <x, y, z>^2_{<\mathbf{A},\mathbf{B},\mathbf{C}>} \} \\
\mathbf{f}(\mathbf{T_1}) &= \{ \{a^{\mathbf{A}}, b^{\mathbf{B}}, c^{\mathbf{C}}\}^1,\ \{x^{\mathbf{A}}, y^{\mathbf{B}}, z^{\mathbf{C}}\}^2 \}. \\
\mathbf{g}_\sigma(\mathbf{f}(\mathbf{T_1})) &= \{<b, a, c>^2_{<\mathbf{B},\mathbf{A},\mathbf{C}>},\ <y, x, z>^1_{<\mathbf{B},\mathbf{A},\mathbf{C}>}\} \\
\mathbf{h}(\mathbf{g}_\sigma(\mathbf{f}(\mathbf{T_1}))) &= << \mathbf{B}, \mathbf{A}, \mathbf{C}>, <y, x, z>, <b, a, c>> \\
\mathbf{h}(\mathbf{g}_\sigma(\mathbf{f}(\mathbf{T_1}))) &= \mathbf{TA}_6
\end{aligned}$$

Notice that $\mathbf{f}(\mathbf{T_1})$ has already been defined by $\mathcal{DC}(\mathbf{T}_i)$. To insure that mappings from Tables to Table-arrays exist and are reversible there needs to exist an equivalent of $\mathcal{DC}(\mathbf{T}_i)$ for Table-arrays.

5.3.2 Mapping Table-Arrays to Tables

Having established a mathematical identity for arrays that represent all the data content of Tables, an operation of the form $\mathcal{TC}(\mathbf{TA}_i) = \mathcal{TC}(\mathbf{TA}_j)$ needs to be defined that asserts when two arrays share the same Table data. Thus, for all appropriate $\mathbf{T}$'s and $\mathbf{TA}$'s, $\mathcal{DC}(\mathbf{T}_i) = \mathcal{TC}(\mathbf{TA}_j)$ must hold. Let $\mathcal{TC}(\mathbf{TA}_j)$ be defined by

$$\begin{aligned}
&\mathcal{TC}(<<D_1, .., D_k>, <v_{1,1}, .., v_{1,k}>, .., <v_{r,1}, .., v_{r,k}>>) \\
&\quad = \{ \{ v^{D_1}_{1,1}, .., v^{D_k}_{1,k} \}, .., \{ v^{D_1}_{r,1}, .., v^{D_k}_{r,k} \} \}.
\end{aligned}$$

In the mappings of Figure 10 from Tables to Table-Arrays $\mathbf{f}(\mathbf{T_i})$ equaled $\mathcal{DC}(\mathbf{T}_i)$ thereby extracting the data content from Tables in USR environments. Similarly, in Figure 11, $\mathbf{h}'(\mathbf{TA_j})$ is set equal to $\mathcal{TC}(\mathbf{TA}_j)$. It is also assumed that $\mathcal{TC}(\mathbf{TA}_j) = \mathcal{DC}(\mathbf{T}_i)$.

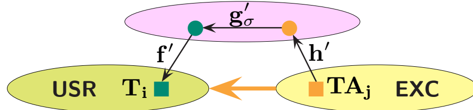

Figure 11: Content Preserving Array-Table Mappings

Again assuming required functions exist, then mapping $\mathbf{TA_8}$ to $\mathbf{T_9}$ might proceed as follows with $\sigma = <C, A, B>$:

$$\begin{aligned}
\mathbf{TA}_8 &= << \mathbf{B}, \mathbf{C}, \mathbf{A}>, <y, z, x>, <b, c, a>> \\
\mathbf{h}'(\mathbf{TA_8}) &= \{ \{y^{\mathbf{B}}, z^{\mathbf{C}}, x^{\mathbf{A}}\}^1,\ \{b^{\mathbf{B}}, c^{\mathbf{C}}, a^{\mathbf{A}}\}^2 \}. \\
\mathbf{g}'_\sigma(\mathbf{h}'(\mathbf{TA_8})) &= \{<z, x, y>^2_{<\mathbf{C},\mathbf{A},\mathbf{B}>},\ <c, a, b>^1_{<\mathbf{C},\mathbf{A},\mathbf{B}>}\} \\
\mathbf{f}'(\mathbf{g}'_\sigma(\mathbf{h}'(\mathbf{TA_8}))) &= << \mathbf{C}, \mathbf{A}, \mathbf{B}>, <c, a, b>, <z, x, y>> \\
\mathbf{f}'(\mathbf{g}'_\sigma(\mathbf{h}'(\mathbf{TA_8}))) &= \mathbf{T}_9
\end{aligned}$$

The combined collection of functions used for supporting data content sharing between Tables and arrays qualify as a formal application data bridge mechanism required as a component to a formal RDM system data model[17].

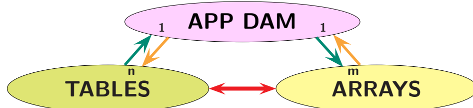

Figure 12: Table/Array Application DAM

A formal modeling of data content sharing between Tables and arrays can be mathematically supported using established extended set operations[18]. This completes the first component of an RDM system data model, a formal modeling of application data bridge mechanisms equating Table data content with array data content.

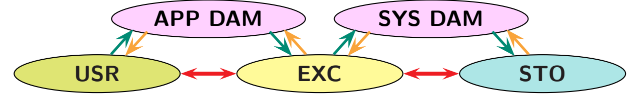

Figure 13a: Components of an RDM System Data Model

The second component, a formal modeling of system data access bridge mechanisms for extracting relevant application data from massive amounts of repository data, is somewhat more challenging.

6. SYSTEM DAMS

System data access mechanisms provide key performance support for application data access needs. The dominant performance consideration is rapid response to application requirements. Examination of traditional record-oriented data access mechanisms suggests that there is a potential performance improvement on the order of three magnitudes.

6.1 Record-Oriented DAMs

A *record*[19] is the traditional unit for data representation and manipulation. Records are conceptually convenient and amenable to mechanical modeling. Since arrays of records are easily processed by programs, a reasonable strategy for *loading* a repository is to store arrays as files. However, accessing relevant data[20] from a record-oriented repository efficiently is severely constrained.

Lack of data independence[21] imposes irresolvable design constraints on system developers. The only solution, in order to improve performance, is to remove the constraints by supporting data independence between applications and data repositories.

6.2 DAM Data Independence

Assuming that data independent DAMs could provide a significant performance difference and before expending potentially wasted effort, it might be prudent to explore what constraints could be removed and what possible affect that might have on system performance.

When application records are stored directly as repository records[22], it is very difficult to achieve higher than a 10%[23] informationally dense I/O transfer. Relieving this constraint could provide a tenfold performance improvement.

Since mechanical access mechanisms need to allow buffer space for updating data, I/O buffers are typically only half full of repository data. Removing this constraint could improve I/O throughput by a factor of two.

As was shown in Figure 3, using I/O buffers of 32 MB is 10 times faster than using I/O buffers of 64 K, and well over 50 times faster than using 8 K I/O buffers. Just switching to 32 MB I/O buffers gives a ten to over fifty-fold improvement

[16] see [Ch05] for examples
[17] Figures 6 & 7
[18] [Ch86, Ch05, Ch07]
[19] a sequence of values stored in consecutive locations.
[20] containing just those records required by an application
[21] discussed earlier in relation to application knowledge of repository data organization.
[22] as is typical with mechanical data models
[23] Stonebraker [St05+] slide 7

in I/O performance. Removing all three constraints gives a potential performance improvement worth exploring.

6.3 DAM Constraints

Traditional data access mechanisms constrain I/O performance by sharing record structure knowledge between applications and repositories. As presented earlier there are $n!$ different concrete representations of an n-arry data relationship. If one ordering is preferable for execution and another is preferable for repository organization and access, then any choice is going to impede system performance.

Sharing a concrete data organization between applications and repositories is as devastating for I/O performance as it is conceptually convenient for system developers. Both the traditional row-store and the recent column-store repository support for record-oriented DAMs unduly inhibit potential system performance and require heroic development efforts from system architects.

As in removing the structural dependence for application data exchange between Table and arrays using the abstract structure of a Table-array, the same approach needs to be taken in removing the structural dependence for system data exchange between application arrays and repository files using the abstract structure of a *Table-Set*,

7. TABLE-SETS

Table-arrays were defined as the unit of manipulation for application data management. Similarly, Table-sets are defined as the unit of manipulation for system data access.

Table-arrays provide an abstract, structure independent representation of data content shared between Tables and arrays. Similarly, Table-sets provide an abstract, structure independent representation of data content between arrays and files.

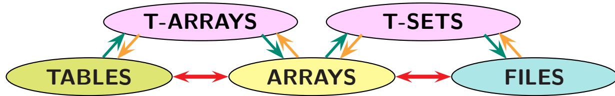

Figure 13b: Table-Sets for Structure Independent Data Access

The key contribution of Table-sets is to provide data independent access to application array data preserved in a repository.

7.1 Table-Sets as Sets

Table-Sets, or T-sets, are nothing more then collections of data items that have a well-defined set theoretic description and a data content preserving representation of arrays as files in a data repository.

Since there are no restraints on the formal expression of Table-sets, all that is necessary is to pick one that covers all possible concrete arrays of interest and that does not incur any structural baggage when transformed into a concrete file representation.

This representation freedom can best be exploited by only limiting the set representation of Table-sets as required by limitations inversely imposed by concrete array and concrete file representation constraints.

7.2 Arrays as Cell Structures

Every 2-dimensional array is an ordered collection of $i \times j$ cells. Just three variables uniquely determine such an array: row location, r_i, column location, c_j, and value, $v_{i,j}$.

Using row locations and column locations as unique labels allows any concrete array to be represented as a labeled array as in Figure 14. Uniquely labeled arrays, in general, are not restricted to two dimensions, nor to location labels, nor even to non-empty cell locations.

The only qualifying criteria for a uniquely labeled array is that all the labels for each specific dimension be unique.

ULA	c_1	c_2	$\ldots$	c_{m-1}	c_m
r_1	$v_{1,1}$	$v_{1,2}$	$\ldots$	$v_{1,m-1}$	$v_{1,m}$
r_2	$v_{2,1}$	$v_{2,2}$	$\ldots$	$v_{2,m-1}$	$v_{2,m}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
r_{n-1}	$v_{n-1,1}$	$v_{n-1,2}$	$\ldots$	$v_{n-1,m-1}$	$v_{n-1,m}$
r_n	$v_{n,1}$	$v_{n,2}$	$\ldots$	$v_{n,m-1}$	$v_{n,m}$

Figure 14: Uniquely Labeled Array **ULA**.

Using **ULA** as defined by Figure 14, a first pass at a set-theoretic definition for a Table-set, $\mathbf{TS}_{\mathbf{ULA}}$, is given by:
$\mathbf{TS}_{\mathbf{ULA}} = \{ \{v_{i,j}^{<r_i,c_j>}\} : \textit{with } v_{i,j},\ r_i,\ \textit{and } c_j \textit{ from } \mathbf{ULA} \}$.

Since Table-arrays have already been shown to map to concrete application arrays, all that is required of the above definition at this point is that it is content compatible with Table-arrays.

A1	*c1*	*c2*	*c3*
r1	*a*	*b*	*c*
r2	*x*	*y*	*z*

A6	*c2*	*c1*	*c3*
r2	*y*	*x*	*z*
r1	*b*	*a*	*c*

A12	*c3*	*c2*	*c1*
r2	*z*	*y*	*x*
r1	*c*	*b*	*a*

Figure 15: Uniquely Labeled Arrays, **A1**, **A6**, **A12**.

Artificially (but consistently) labeling the content equivalent arrays of Figure 8 produces the three uniquely labeled arrays of Figure 15. All three of these arrays can be expressed by the same set-theoretic expression as:

$$\mathbf{TS}_{\mathbf{A}} = \{ \{a^{<r_1,c_1>}\}, \{b^{<r_1,c_2>}\}, \{c^{<r_1,c_3>}\}, \{z^{<r_2,c_3>}\}, \{y^{<r_2,c_2>}\}, \{x^{<r_2,c_1>}\} \}.$$

By equating $c_1 = A$, $c_2 = B$, and $c_3 = C$, and associating $< A, B, C >, r_1 = 1, r_2 = 2$ with **T1**, $< B, A, C >, r_1 = 1, r_2 = 2$ with **T6**, and $< C, B, A >, r_1 = 1, r_2 = 2$ with **T12** the content equivalent RDM-Tables of Figure 16 are easily derived.

T1	A	B	C
	a	*b*	*c*
	x	*y*	*z*

T6	B	A	C
	y	*x*	*z*
	b	*a*	*c*

T12	C	B	A
	z	*y*	*x*
	c	*b*	*a*

Figure 16: Unique RDM-Tables, **T1**, **T6**, **T12**.

The RDM-Tables of Figure 16 are a subset of RDM-Tables of Figure 9 which were shown mappable to Table-arrays. Since Table-sets are adequate representations of Table-arrays, it just remains to show a similar compatibility between Table-sets and files.

7.3 Table-sets as Files

If *files* are considered to be any repository-resident collection of addressable concrete objects, then an abstract representation of a file's content only has to capture the

mathematical identity of the addressable objects. Though the addressing mechanism is important for performance, it is an artificially imposed data structuring and not an intrinsic attribute of the object being accessed.

Traditionally files[24] are already recognized as sets, but without exploiting the mathematical properties of sets or their elements. A *set-store* file organization preserves the traditional view of files, but expands the data accessing capabilities to exploit set-theoretic advantages.

Since Table-sets and files are both recognized as sets, the only set-store requirement left to mathematically equate them is their content.

7.4 App-Records & Sys-Records

Traditional mechanical data models embed application records into system repository records[25]. It is precisely this embedding of application record content and structure into an accessible unit of repository storage that encumbers traditional data access performance.

The relevant data[26] granularity of a record access is too coarse when only 5% to 10% of the record may be required. Since any combination of the record values may, at any one time, qualify as relevant data, the ideal unit of data access would be individual record values. This fine an access granularity necessitates having data coalescing operations that can dynamically construct collections of highly relevant data records[27] from collections of primitive data elements.

One means for doing this is to equate files with an existing structure that already supports both these requirements.

7.5 Cell-Blocks as Files

Define a *cell* to be a primitive unit of data access from a storage repository. Define a *cell-block* to be a concrete collection of cells. So far, these definitions do not deviate from tradition, since cells can be equated to system records and cell-blocks equated to files.

Earlier it was noted that a 2-dimensional array is an ordered collection of $i \times j$ cells, uniquely determined by: row location, r_i, column location, c_j, and value, $v_{i,j}$.

Given an appropriate graphical representation of a cell, a collection of cells can be equated to the uniquely labeled array of Figure 14 as represented by Figure 17.

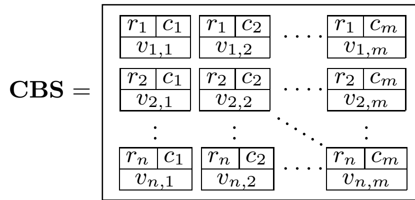

Figure 17: Array Equivalent Cell-Block Structure

The assertion that the cell-block depicted by Figure 17 is mathematically equivalent to the uniquely labeled array of Figure 14 may not be immediately obvious. Even if it is, a mathematical justification is required.

[24] A *file* is a set of similarly constructed records...[LTN p.7].

[25] A *logical record* (or record) is a named collection of data items or attributes treated as a unit by an application program. In storage, a record includes the pointers and record overhead needed for identification and processing by the database management system. [LTN p.7].

[26] data immediately required by an application

[27] containing 90+% relevant data

It needs to be shown that an unstructured collection of cells in a cell-block is mathematically equivalent to an ordered collection of cells of a uniquely labeled array. Doing so also shows that files that are content equivalent to uniquely labeled arrays can be represented by unstructured collections of cells, and thus equivalent to Table-sets.

7.6 Cell-Blocks as Table-Sets

To define any collection of items to be a set first requires the items to be mathematically well-defined. For collections of cells, the cell needs a mathematical identity.

r	c
v	

$$= \{\ v^{<r,c>}\ \}$$

Figure 18: Mathematical Identity of a Cell

With cells defined as sets and since cell blocks are defined as unordered collections of cells, cell-blocks themselves can be defined as sets. For example:

$$\mathbf{S}_{\mathbf{CBS}} = \{\ \{v_{i,j}^{<r_i,c_j>}\} :\ v_{i,j},\ r_i,\ and\ c_j\ in\ cell\ of\ \mathbf{CBS}\ \}.$$

Notice that the above set representation of the cell-block, **CBS**, is identical to the Table-set representation of a uniquely labeled array, **ULA**: $\mathbf{S}_{\mathbf{CBS}} = \mathbf{TS}_{\mathbf{ULA}}$.

With cell blocks having a set-theoretic representation all the operations and properties defined for sets can now be applied to cell blocks, and thus defined for file data content independent of any specific implementation or repository data organization.

This set-theoretic data exchange between applications and repositories is the key contribution of a mathematical model in providing data independent I/O.

8. DATA INDEPENDENT I/O

Given the assertion that a set-theoretic data exchange between applications and data repositories supports data independent I/O does not demonstrate that such is the case.

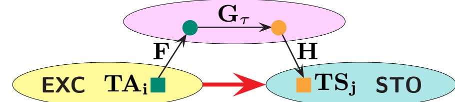

Figure 19: Mapping Array Data to Files

Figure 19 shows that from going from arrays to files, concrete arrays are formally represented as T-arrays, $\mathbf{TA}_i$, concrete files are represented as T-sets, $\mathbf{TS}_j$, and there exist functions giving: $\mathbf{H}(\mathbf{G}_\tau(\mathbf{F}(\mathbf{TA}_i))) = \mathbf{TS}_j$. Where:

$\mathbf{F}(\mathbf{x})$ is a structure preserving set representation of $\mathbf{x}$,
$\mathbf{G}_\tau(\mathbf{y})$ abstracts row-ordering structure of $\mathbf{y}$, and
$\mathbf{H}(\mathbf{z})$ is a structure neutral file representation of $\mathbf{z}$.

Assume set-theoretic functions have been defined for the mapping of Tables-Arrays to Table-Sets, then mapping $\mathbf{TA_6}$ to $\mathbf{TS_1}$ might proceed as follows with $\tau = \emptyset$:

$$\mathbf{TA}_6 \ = << \mathbf{B}, \mathbf{A}, \mathbf{C} >, < y, x, z >, < b, a, c >>$$

$$\mathbf{F}(\mathbf{TA_6}) \ = \ \{< b, a, c >^2_{<\mathbf{B},\mathbf{A},\mathbf{C}>},\ < y, x, z >^1_{<\mathbf{B},\mathbf{A},\mathbf{C}>}\}$$

$$\mathbf{G}_\tau(\mathbf{F}(\mathbf{TA_6})) \ = \ \{\ \{a^{\mathbf{A}}, b^{\mathbf{B}}, c^{\mathbf{C}}\}^{r_2},\ \{x^{\mathbf{A}}, y^{\mathbf{B}}, z^{\mathbf{C}}\}^{r_1}\ \}.$$

$$\mathbf{H}(\mathbf{G}_\tau(\mathbf{F}(\mathbf{TA_6}))) \ = \ \{\ \{y^{<r_1,\mathbf{B}>}\}, \{x^{<r_1,\mathbf{A}>}\}, \{z^{<r_1,\mathbf{C}>}\}, \{b^{<r_2,\mathbf{B}>}\}, \{a^{<r_2,\mathbf{A}>}\}, \{c^{<r_2,\mathbf{C}>}\}\ \}.$$

$$\mathbf{H}(\mathbf{G}_\tau(\mathbf{F}(\mathbf{TA_6}))) \ = \ \mathbf{TS}_1$$

Though the above mapping of Table-arrays to Table-sets confirms the existence of a mathematical mechanism for entering data into a storage repository. It does not, however,

show any advantage to over the WYSIWYG architecture of entering data in the way it is expected to be retrieved.

For an architecture to support informationally dense I/O performance, the reverse mechanism of extracting data has to be able to select just units of relevant data, independent of how they were entered.

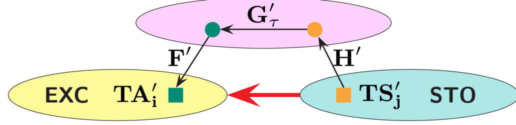

Figure 20: Extracting Relevant Array Data from Files

Figure 20 shows the transforming of file data to array data, formally represented as T-arrays, $\mathbf{TA}_i$, concrete files are represented as T-sets, $\mathbf{TS}_j$, and there exist functions giving: $\mathbf{F}'(\mathbf{G'}_\alpha(\mathbf{H}'(\mathbf{TS}_j))) = \mathbf{TA}_i$. Where:

$\mathbf{H}'(\mathbf{x})$ transmutes elements of Table-set $\mathbf{x}$,
$\mathbf{G}'_\alpha(\mathbf{y})$ adds element and domain orderings to $\mathbf{y}$, and
$\mathbf{F}'(\mathbf{z})$ transforms $\mathbf{z}$ to a Table-array.

Again assuming required functions exist, then mapping $\mathbf{TS_2}$ to $\mathbf{TA_1}$ might proceed as follows with $\alpha = <A, B, C>$:

$$\begin{aligned}\mathbf{TS}_2 \;&=\; \Big\{\ \{b^{<s_1,\mathbf{B}>}\}, \{a^{<s_1,\mathbf{A}>}\}, \{c^{<s_1,\mathbf{C}>}\},\\ &\qquad \{y^{<s_2,\mathbf{B}>}\}, \{x^{<s_2,\mathbf{A}>}\}, \{z^{<s_2,\mathbf{C}>}\}\ \Big\}.\\ \mathbf{H}'(\mathbf{TS_2}) \;&=\; \Big\{\ \{b^{\mathbf{B}}, c^{\mathbf{C}}, a^{\mathbf{A}}\}^{s_1},\ \{y^{\mathbf{B}}, z^{\mathbf{C}}, x^{\mathbf{A}}\}^{s_2}\ \Big\}.\\ \mathbf{G}'_\alpha(\mathbf{H}'(\mathbf{TS_2})) \;&=\; \Big\{< a, b, c >^1_{<\mathbf{A},\mathbf{B},\mathbf{C}>},\ < x, y, z >^2_{<\mathbf{A},\mathbf{B},\mathbf{C}>}\Big\}\end{aligned}$$

$$\mathbf{F}'(\mathbf{G}'_\alpha(\mathbf{H}'(\mathbf{TS_2}))) = << \mathbf{A}, \mathbf{B}, \mathbf{C} >, < a, b, c >, < x, y, z >>$$
$$\mathbf{F}'(\mathbf{G}'_\alpha(\mathbf{H}'(\mathbf{TS_2}))) \;=\; \mathbf{TA}_1$$

The above set operation examples demonstrate that data exchange between applications and repositories can be controlled mathematically without any knowledge of the physical organization of repository data. This establishes the feasibility of data independent set-store architectures, but gives absolutely no clue as to how it promotes superior data access from concrete file structures.

8.1 Set-Store File Architectures

File structures are denoted by their unit of data access. Traditional row-store and column-store architectures rely on mechanical data models for DAM implementations to locate and return appropriate collections of rows or columns.

Though not explicitly disclosed above, the set processing operations performed by $\mathbf{H}'$ require the unit of data access for file structures to be a set. Since files are concrete objects, it is not always apparent what its mathematical identity might be as an abstract set.

8.2 Files as Cell-Blocks

Earlier in this paper cell-blocks were introduced to help visualize the transition from arrays to file. With added structure, cell-blocks can be used to reflect the set-theoretic essentials of different file structures.

Though existing row-store and column-store architectures are traditionally mechanically modeled, they can also be modeled mathematically by set-store architectures. All this requires is a well-defined abstract set representation for concrete row and column file structures[28].

[28] including operations for supporting a set-theoretic DAM

8.3 Row-Store Files

Earlier a *cell-block* was defined to be a concrete collection of cells. This definition can now be expanded to include cell-blocks of cell-blocks.

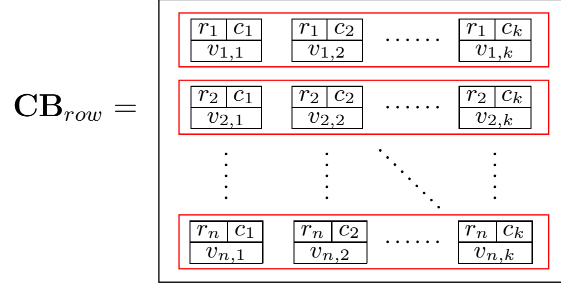

Figure 21: Cell-Block of Row Structures

The graphic of Figure 21 is neither a mathematically well-defined set nor a machine acceptable file structure. It is just a visual aid that, properly interpreted, can provide conceptual assurance that certain sets and certain files have the same *meaning*. Some set expressions that capture the essential properties are:

$$\mathbf{S}_{\mathbf{row}} = \big\{\ \{v_{1,1}^{c_1}, .., v_{1,k}^{c_k}\}^1, ..., \{v_{n,1}^{c_1}, .., v_{n,k}^{c_k}\}^n\big\}.$$
$$\mathbf{F}_{\mathbf{row}} = << c_1, .., c_k >, < v_{1,1}, .., v_{1,k} >, ..., < v_{n,1}, .., v_{n,k} >>$$

For $r_i = i$ all three of the above can be argued to share the same data relationships, but with different representation structurings.

8.4 Column-Store Files

Using the expanded definition for cell-blocks, column-store file representations can also be visually displayed.

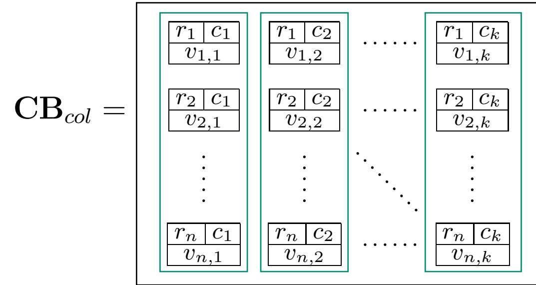

Figure 22: Cell-Block of Column Structures

The graphic of Figure 22 can also be used as a visual aid to assure that certain sets and certain files have the same *meaning*. Again, set expressions that capture the essential properties are:

$$\mathbf{S}_{\mathbf{col}} = \big\{\ \{v_{1,1}^1, .., v_{n,1}^n\}^{c_1}, ..., \{v_{1,k}^1, .., v_{n,k}^n\}^{c_k}\big\}.$$
$$\mathbf{F}_{\mathbf{col}} = << c_1, v_{1,1}, .., v_{n,1} >, ..., < c_k, v_{1,k}, .., v_{n,k} >>$$

Again, for $r_i = i$ all three of the above can be argued to share the same data relationships, but with very different representation structurings.

The fact that file structures can be modeled as sets is the key factor for implementing data independent I/O access mechanisms[29].

[29] though presumably not impossible using mechanical data models, its hard to imagine it becoming a general practice.

Not yet established, however, is exactly what benefit, if any, data independence and set processing at the I/O level achieve, nor what consequences there may be imposed on *updating*[30] the repository.

Though not explicitly disclosed above, the set processing operations performed by $\mathbf{H}'$ dictate I/O performance. $\mathbf{H}'$ actually represents a family of simultaneous set operations working on multiple input Table-sets. These operations[31] are controlled by the system DAM which governs all data representation and organization in the repository.

9. DAM I/O

Common practice in the industry is to advocate I/O avoidance when system performance is a primary concern. This is a legitimate position for systems functioning under mechanical data model design constraints, but just the opposite is true for systems functioning under the benefits of a mathematical data modeling.

By employing mathematical modeling and use of parallel I/O paths, PIOPs, I/O performance is only limited by the speed and size of CPUs and RAM available. It has been argued that this claim can be achieved using a data access mechanism, DAM, that takes full advantage of data independent I/O potential between application data and system repository data.

Such a DAM I/O architecture must support massively[32] parallel, informationally dense I/O data transfers. This is only achieved by a data repository architecture supporting dynamic data partitioning[33] and data reorganization.

9.1 Informationally Dense I/O

For optimized I/O data transfers a DAM has to recognize and access those *pieces*[34] of files that support the application requirement minimum, ARM, from repository data.

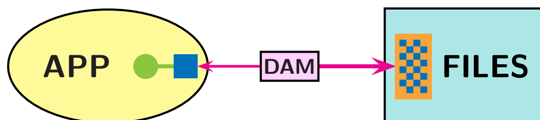

Figure 23. Informationally Dense I/O Data Transfers.

In Figure 23, abstract Tables are represented by (●), abstract Table-arrays are represented by (■), and concrete files are represented by (▮).

In a traditional DAM architecture all the above pieces would have already been determined during an intensive database design phase. It would have been decided what collections of data would most likely be required by an application and which of those collections would have been indexed and distributed in a repository for best predictive anticipated application requirements.

[30] since all physical data representations are considered as sets and since sets are immutable, no changes are EVER made to physical sets. New data is added as a set and integrated through set operations. Old data is always kept, but depreciated in favor of more current values or relationships.

[31] all RDM operations plus set operations are available here

[32] 24+ parallel I/O paths by today's standards

[33] and non-disjoint decompositions

[34] little blue squares in orange block, files, in Figure 23

9.2 Traditional DAMs

Traditional mechanical data models have to consider all repository data organizations not only as just best to serve application data access response times, but also to support non-disruptive and reliable updating of data values and data relationships.

9.3 Mathematical DAMs

Since with mathematical data models updating is always a constructive[35] process and no modifications to existing data are required[36], the only database design concern is to make sure all data required to support application data access requirements, is that all the data is actually *loaded*[37]

9.4 DAM Strategies

This is not meant to imply that use of mathematical data models do not require considerable work and design intelligence to support high performance I/O data access, but only that the effort is less predictive and more adaptive in nature.

In an example to follow using a killer query from the industry standard TPC-H benchmark, knowledge of Table Domains relevant to each query can provide a first pass at the organization of repository Table-sets best tailored for informationally dense I/O for each query.

Since files are only visible to applications as Table-sets, the decision of what files and with what data organization are to be made available to applications, is really just a question of what sets can be determined to be the most informationally dense.

10. ANATOMY OF A QUERY

The TPC-H benchmark has recently demonstrated the performance advantages of using column-store repository architectures over row-store architectures. Since these are both based on mechanical data models for data access, they both lack data independent interfaces between application processing data and repository data representations. Thus, neither can take full advantage of the performance potential of their host platform.

10.1 TPC-H Benchmark Tables

Data relationships are unique. The number of RDM-Table representation for a unique collection of data relationships is a factorial function of the number of RDM Domains. The TPC-H LINEITEM[38] Table has 16 Domains and thus 16! distinct RDM-Table representations. By casting the LINEITEM Table as a set, only one representation need be considered by a programmer.

Since the LINEITEM Table has 16 Domains, no logic is lost by referring to them as L1,..,L16. Then a specific LINEITEM Table can be defined as a set by:

$$\mathbf{L} = \{\{\ x_1^{L1},\ x_2^{L2}, .., \ x_{16}^{L16}\ \}^{r_i} : < x_1, ., x_{16} > \ \in LINEITEM\ \}$$

For the 1TB benchmark $\mathbf{L}$ has a cardinality of 6,000,000,000

[35] no existing data or data relationships are destroyed.

[36] nor even allowed.

[37] loaded here means only that the data is available, not that traditional mechanical access structures have been constructed.

[38] see [TPC] for detailed description

and file size of 750 GB. Certainly not a set that one would often want to slog back and forth from disks.

Since none of the 22 TPC-H queries requires all the Domains of **L** only the ARM of a query needs to be accessed from disk. In the case of Query 9 only 7 Domains are required.

$$\mathbf{L_{Q9}} = Ɖ_{\{L1,L2,L3,L4,L5,L6,L7\}}(\mathbf{L})$$

With sufficient foresight $\mathbf{L_{Q9}}$ could be generated and stored at load time, taking only 125 GB of disk space and setting up an informationally dense I/O access for the execution of subsequent Query 9's.

In order to get the content of $\mathbf{L_{Q9}}$ properly ensconced in a data repository, first requires transforming a purely symbolic representation into a concrete array representation. The next requirement is to map 1 formal expression of data content to m structurally distinct array objects.

Following is an example of using set-store data repositories for loading TPC-H data for optimal I/O performance for each of the 22 queries, and the actual I/O requirements for executing query 9.

SQL EXPRESSION OF TPC-H Query 9:

```
SELECT < NATION, Oyear, SumProfit >
 FROM SELECT < NATION, o_year, amount >
       FROM  L, O, H, P, S, N
       WHERE [P2] = LIKE '%[color]%'
       WHERE [P1] = [L2]
       WHERE [H1] = [L2]
       WHERE [H2] = [L3]
       WHERE [O1] = [L1]
       WHERE [S1] = [L3]
       WHERE [S4] = [N1]
```

WHERE statements are mapped to *scope conditionals* in XST. By **[Q_Q2] = [L_L3]**, it is meant that the value of an element in an element of **Q** with a scope of **Q2** is equal to the value of an element of an element of **L** with a scope of **L3**. For a complete description of this analysis see [Ch09a].

Execution of TPC-H Query 9:

```
[0] Open_Table_Set(L{L1,L2,L3,L4,L5,L6,L7})
    Open_Table_Set(O{O1,O5})
    Open_Table_Set(H{H1,H2,H4})
    Open_Table_Set(P{P1,P2})
    Open_Table_Set(S{S1,S4})
    Open_Table_Set(N{N1,N2})
[1] Get_Table_Array(P<P1,P2>)                        11,800,000*Sf IN
    App_SetOP(Like: P2=green, RS_1 )
    Put_Table_Array(RS_1<P1>)                            46,612*Sf OUT
[2] Get_Table_Array(RS_1<P1>)                            46,612*Sf IN
    Get_Table_Array(H<H1,H2,H4>)                      9,600,000*Sf IN
    App_SetOP(Restrict: H1=P1, RS_2 )
    Put_Table_Array(RS_2<H1,H2,H4>)                     558,576*Sf OUT
[3] Get_Table_Array(RS_2<H1,H2,H4>)                     558,576*Sf IN
    Get_Table_Array(L<L2,L3,L1,L4,L5,L6,L7>)        132,000,000*Sf IN
    App_SetOP(Join: <H1,H2>=<L2,L3>, RS_3 )
    Put_Table_Array(RS_3<L1,L4,L2,L3,L5,L6,L7,H4>) 9,067,760*Sf OUT
[4] Get_Table_Array(RS_3<L1,L4,L2,L3,L5,L6,L7,H4>) 9,067,760*Sf IN
    Get_Table_Array(O<O1,O5>)                        12,000,000*Sf IN
    ADF_1(O: Oyear)
    ADF_2(RS_3: Amount)
    App_SetOP(Join: <L1>=<O1>, RS_4 )
    Put_Table_Array(RS_4<L3,Oyear, Amount>)           3,467,600*Sf OUT
[5] Get_Table_Array(RS_4<L3,Oyear, Amount>)           3,467,600*Sf IN
    Get_Table_Array(S<S1,S4>)                            50,000*Sf IN
    Get_Table_Array(N<N1,N2>)                               659 IN
    ADF_3(RS_4: SumProfit)
    App_SetOP(Join: <L3>=<S1>, RS_5 )
    Save_Table_Set(RS_5<N2,Oyear, SumProfit>)               175 OUT

                              TOTAL I/O = 191,731,096*Sf
```

Note: Get_Table_Array(P<P1,P2>) implies from Table_Set **P**.

As was described earlier, for an SQL-friendly XSP engine, the interface algebra must provide a functional covering for SELECT statements. By definition, an XSP engine is any implementation of a set-processing algebra closed under extended set membership. Thus, internal representations of sets can not be known and any collection of interface operations is admissible. Any two XSP engines can be interchanged as long as they are functionally equivalent and the interfaces have a known correspondence. The only discernible difference between any two XSP engines is their performance. (An XSP engine implementation could even have a structure-dependent indexed-record access internal architecture. No one would ever be able to tell.)

11. SYSTEM PERFORMANCE

The only difference between past, present, and potential systems is the degree of data independence supported by the system architecture. The architecture of present systems has not changed for thirty years. Performance of present systems is increasingly handicapped by forcing application processing environments to be cognizant of repository data representations and access organizations. It is sometimes claimed that more application processing resources are devoted to *finding data* than to *processing data*.[39] There is no reason not to assume that if application processing environments were designed to do 99.9% useful work, and given enough CPU power to keep up with optimized I/O data access, that performance would only be limited by I/O throughput.

11.1 Platform Performance Potential

Given a platform with n-PIOPs[40] and assuming enough processing power to keep up with I/O throughput, a lower bound on system performance can be established by the time it takes to fully load all relevant data and to execute applications of interest.

Using the TPC-H results[41] for data loading and Query 9 execution times for 1TB of data, a platform performance comparison can be made between actual results and those possible by utilizing I/O throughput.

TPC-H 1000GB 11/01/09			
System	**Full Load**	**Query 9**	**TET**
24-PIOP	4.67 min	24.36 sec	5.08 min
48-PIOP	2.34 min	12.18 sec	2.54 min
64-PIOP	1.75 min	9.14 sec	1.90 min
96-PIOP	1.17 min	6.09 sec	1.27 min
Oracle(48)	142.80 min	24.3 sec	143.21 min
IBM(64)	49.00 min	500.7 sec	57.35 min
ParAccel(96)	41.30 min	32.4 sec	41.84 min
Microsoft(97)	354.00 min	215.7 sec	357.59 min

Figure 30: Performance Potential of I/O Throughput

Though the TPC-H benchmark does not consider load times to be relevant in the calculation of the performance metrics, they can be the cause of lost opportunity in real commercial situations.

[39] useful processing of OLTP data less than 7% [Ha08+], suggests OLAP useful processing possibly as high as 25%.
[40] parallel I/O paths
[41] as of 11/01/09

11.2 Loading for I/O Performance

Since the minimal platform has only 48 parallel I/O ports, the best potential load time would be 2.34 minutes. The best reported load time is 17 times the best possible load time. The worst reported load time is 151 times the best possible load time, for a 48-PIOP system. The reported load time on a 97-PIOP platform gives a staggering load time 302 times that of the platform potential.

Since different platforms have different I/O throughput potentials, a better comparison of platform performance potential utilization would be to compare actual system execution times with matching PIOP possibilities.

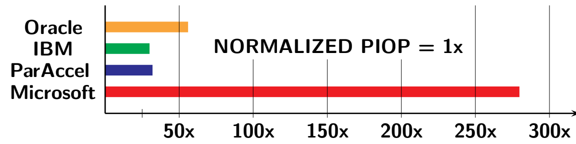

Figure 31: Deviation from Platform Potential

Figure 31 shows how much more time is actually being consumed than is required by proper utilization of I/O throughput. IBM and ParAccel are the closest by being 30 and 32 times off the optimal[42]. Oracle, though it has the best Query 9 time, is still off the mark by a factor of 56. Microsoft deviates from the optimal by a factor of 281.

11.3 Query Execution Performance

In cases where data loading performance is not a critical concern but where repeated queries are accessing pre-loaded data, the performance shifts to pure execution comparisons.

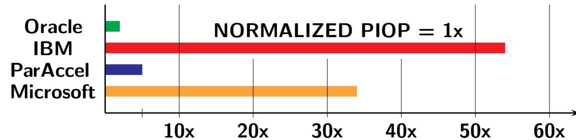

Figure 32: Query 9: Deviation from Optimal Performance

11.4 TET System Performance

Since real commercial installations are using computers to gain a competitive advantage, the real cost to a business is not the cost of equipment but the cost of lost business. Spending two or three million more dollars in three years on equipment to save from losing two or three million dollars a quarter is of more interest to business than a geometric mean of queries per hour. Only one real issue faces businessmen concerned with competitive advantage, the total elapsed time, TET, required to keep ahead of the competition.

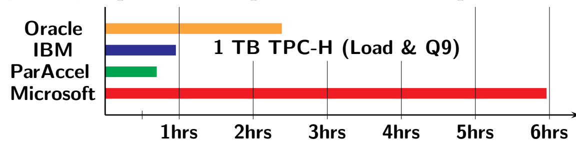

Figure 33: TET: Time Critical Productivity Comparison

12. CONCLUSION

What may be the greatest deficiency in the development of software systems is the lack of *any* formal data model for specifying application intent, translating that intent into executable routines, and supporting preserved data for access by those routines.

The first part of this paper analyzes the I/O throughput potential of existing hardware platforms. Just by using 32 MB I/O buffers instead of 4 KB I/O buffers system I/O throughput can be improved by a factor of 150. By also improving information density[43] from 10% to 80% gives over three magnitudes of performance improvement on a single I/O path. By providing multiple parallel I/O paths[44] I/O throughput can dramatically improve overall system performance.

The second part of the paper argues that the use of mechanical data models forces system developers to avoid use of I/O at all costs. Structurings imposed on data for best execution are seldom best structurings for the preservation and rapid access of repository data. Without the ability to independently separate and manage processing structures from preservation and access structures system performance potential is severely compromised.

By using an extended form of set-theoretic notation it is shown that the mathematical identity of data can be used to isolate data content from data structuring and manage each one separately. This, in turn, allows system developers to pick the best structuring for processing and the best structuring for preservation and data access.

Since academic arguments are sometimes hard to fully appreciate, but are always easy to completely dismiss, a well known industry benchmark[45] was chosen to provide a backdrop for a performance analysis and actual execution[46].

In summary, mathematical control of the representation, organization and manipulation of data can allow developers of future systems to approach the performance potential of any given hardware platform. With such control and given enough CPU power for applications to keep up with optimized I/O data access, there is no reason to assume that system performance could not be arbitrarily improved by parallel I/O throughput.

13. REFERENCES

[Ba73] Bachman, C.: *The Programmer as Navigator*, 1973 ACM Turing Award Lecture. Comm. ACM 16, 11 (November 1973), 653-658.
http://www.jdl.ac.cn/turing/pdf/p653-bachman.pdf

[Cha01] Champion, M.: *XSP: An Integration Technology for Systems Development and Evolution*, Software AG - 2001
http://xsp.xegesis.org/Xsp-uxr.pdf

[Ch77] Childs, D. L.: *Extended Set Theory: A General Model for Very Large, Distributed, Backend Information Systems*, Third International Conference On Very Large Databases, Tokyo, Japan, 1977
http://xsp.xegesis.org/VLDB_77abstract.pdf

[Ch86] Childs, D. L.: *A Mathematical Foundation For Systems Development*, NATO ASI Series, Vol F24, Database Machines, Edited by A. K. Sood and A. H. Qureshi, Springer-Verlag, 1986
http://xsp.xegesis.org/Nato_asi.pdf

[Ch05] Childs, D. L.: *Set-Processing at the I/O Level:* A Performance Alternative to Traditional Index Structures, Integrated Information Systems
http://xsp.xegesis.org/Spio.pdf

[42] normalized to compatible PIOPs

[43] transferring application usable data only

[44] PIOPs in the paper

[45] TPC-H data: full data load and execution of Query 9

[46] results confirmed the academic predictions

[Ch07] Childs, D. L.: *Data Representations as Mathematical Objects, Considering Content Compatibility of Relational & XML Data Representations*, Integrated Information Systems - [4/25/07]
http://xsp.xegesis.org/Mi_data.pdf

[Ch09a] Childs, D. L.: *Axioms for an Extended Set Theory* , Integrated Information Systems
http://xsp.xegesis.org/X_axioms.pdf

[Ch09c] Childs, D. L.: *Notes On: Items, Sets, Names, Tuples, & Klasses* , Integrated Information Systems
http://xsp.xegesis.org/N_isntk.pdf

[Co70] Codd, E. F.: *A Relational Model of Data for Large Shared Data Banks*, CACM 13, No. 6 (June) 1970
http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf
http://xsp.xegesis.org/Relations_70.pdf[p. 379-380]

[Ha08+] Harizopoulos, S.; Madden, S.; Abadi, D.; Stonebraker, M.: *OLTP Through the Looking Glass, and What We Found There*, SIGMOD'08, June 9-12, 2008, Vancouver, BC, canada
http://cs-www.cs.yale.edu/homes/dna/papers/oltpperf-sigmod08.pdf

[LTN] Lightstone, S; Teorey, T.; Nadeau, T.: *Physical Database Design*, Morgan Kaufmann, 2007
http://lightstone.x10hosting.com/BookFlyer-physdb.htm

[Sk57] Skolem, Thoralf: *Two Remarks on Set Theory,* Mathematica Scandinavica 5 (1957), p.43-46.
(http://www.mscand.dk/article.php?id=1481)

[St05+] Stonebraker, M.:(many others): *C-Store: A Column-oriented DBMS (conference slides)* By. New England Database Group. M.I.T.
slides: http://www.vldb2005.org/program/slides/thu/s553-stonebraker.ppt
paper: http://db.csail.mit.edu/projects/cstore/vldb.pdf

[St07+] Stonebraker, M.; Madden, S.; Abadi, D.; Harizopoulos, S.; Hachem, N.; Helland, P.: *The End of an Architectural Era (It's Time for a Complete Rewrite)*. 33rd International Conference on Very Large Data Bases, Vienna, Austria, 2007.
http://www.ahzf.de/itstuff/papers/vldb07hstore.pdf

[TPC] TPC-H Specifications & Top Ten
http://www.tpc.org/tpch/spec/tpch2.8.0.pdf
http://www.tpc.org/tpch/results/tpch_perf_results.asp