Here are my Data Files. Here are my Queries. Where are my Results?

Stratos Idreos[†] Ioannis Alagiannis[⋆] Ryan Johnson[‡] Anastasia Ailamaki[⋆]

[†]CWI, Amsterdam [‡]Carnegie Mellon University [⋆]École Polytechnique Fédérale de Lausanne

ABSTRACT

Database management systems (DBMS) provide incredible flexibility and performance when it comes to query processing, scalability and accuracy. To fully exploit DBMS features, however, the user must *define* a schema, *load* the data, *tune* the system for the expected workload, and answer several questions. Should the database use a column-store, a row-store or some hybrid format? What indices should be created? All these questions make for a formidable and time-consuming hurdle, often deterring new applications or imposing high cost to existing ones. A characteristic example is that of scientific databases with huge data sets. The prohibitive initialization cost and complexity still forces scientists to rely on "ancient" tools for their data management tasks, delaying scientific understanding and progress.

Users and applications collect their data in flat files, which have traditionally been considered to be "outside" a DBMS. A DBMS wants control: always bring all data "inside", replicate it and format it in its own "secret" way. The problem has been recognized and current efforts extend existing systems with abilities such as reading information from flat files and gracefully incorporating it into the processing engine. This paper proposes a new generation of systems where the only requirement from the user is *a link to the raw data files*. Queries can then immediately be fired without preparation steps in between. Internally and in an abstract way, the system takes care of selectively, adaptively and incrementally providing the proper environment given the queries at hand. Only part of the data is loaded at any given time and it is being stored and accessed in the format suitable for the current workload.

1. INTRODUCTION

Database systems can only achieve excellent query response times given a number of low-level *designing and tuning* steps, which are a prerequisite for the system to start processing queries. A significant part of the initialization cost is due to loading the data into files or raw storage according to a format specified by the DBMS, and to physically designing and tuning the data set for the expected workload, i.e., creating the proper indices, materialized views, etc. Flat files of raw data are considered "outside" the DBMS, within which data is viewed and massaged in specific formats. Despite that the assumption of control over the data placement and format entails design advantages for the database system, such a restriction prevents the wide adoption of DBMS technology to new application areas, increases the bootstrap time of a new application and leads to systems that either are appropriate for only one scenario or need to be perennially re-tuned.

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9-12, 2011, Asilomar, California, USA.

1.1 Accessing Flat Files

Both the research community and the commercial world recognize the problem and several crucial steps have already been addressed. For example, many commercial and open-source database systems already offer a functionality that allows a database system to immediately query a flat file. The DBMS essentially links a given table of a schema with a flat file and, during query processing, parses data from the flat file on-the-fly. Oracle, for instance, offers an option to have "external tables" while MySQL enables the "CSV engine". As a result, data can be queried without having to explicitly load the raw data into the DBMS. In practice, however, flat files are still "outside" the DBMS as there is no support for indices, materialized views or any other advanced DBMS optimization. Query processing performance is therefore lower when compared to the the performance of queries running on "internal" tables, so the systems mostly offer external flat files as an alternative way for the user to load/copy data into normal DBMS tables rather than for query processing. Consequently, even though current flat file functionality is a significant step forward, the problem is still an open one as there is still a long way to go towards systems that require zero initialization overhead.

1.2 Motivating Example

Typically, scientific databases grow on a daily basis as the various observation instruments continuously create new data. Thus, the optimal storage and access patterns may change even on a daily basis purely depending on the new data, its properties, correlations, as well as the ways that the scientists navigate through the data and the ways their understanding and data interpretation evolve. In such scenarios, no up-front physical design decision can be optimal in light of a completely dynamic and ever-evolving access pattern. Therefore, people often rely on Unix-based tools or on custom solutions in order to achieve bread-and-butter functionality of a DBMS.

Paying a heavy preparation cost for the unknown is prohibitive if needed on a daily basis. Even simply loading the data in the database is a significant investment and delay for large data sets. Alternatively, one can write a quick Awk script and immediately query any part of the data, avoiding all hassle associated with designing schema, loading data and tuning. Here follow a few example reactions from scientists:

1. Why do I have to wait multiple hours for loading and tuning? I am getting another Terabyte of data tomorrow and I just want to quickly find out if the current data is of any interest so that I go ahead and analyze it.

2. Why should I have to load the whole data set? I just need a small part now; I do not know if, or when, I will need the rest.

3. How can I decide what indices to build or whether to use column- or row-stores? I won't understand the data properties until after I've looked at it! Worse, tomorrow could be a different story.

4. How should I know how to set-up a complex DBMS? I am not a computer scientist. Hiring DB experts to set-up and re-tune the system on a daily basis is extremely expensive.

1.3 Contributions

In this paper, we argue towards a new generation of systems that provide a *hybrid* experience between using a Unix tool and a DBMS; all they need as a start-up step is *a pointer to the raw data files*, i.e., the flat files containing the data, for example in CSV format. There is *zero initialization* overhead as with a scripting tool but at the same time *all advanced* query processing capabilities and performance of a DBMS are retained. The key idea is that data remains in flat files. The user can edit or change a file at any time. When queries arrive, the system will take care of bringing the proper data from the file, and it will store it and evaluate it in an appropriate way.

We first demonstrate the initialization overhead of DBMS and the flexibility of using a scripting language. Then, we demonstrate the suitability of a DBMS for data exploration, when a quick glimpse over the data is not the only goal, i.e., when a user wants to repeatedly query the same data parts. Finally, we describe our vision in detail. Various policies are studied regarding how data can be fetched, cached, reused and how this whole procedure can happen on-the-fly, integrated with query processing in a modern DBMS. We provide a prototype implementation and evaluation over MonetDB. Our results clearly show the potential and benefits of the approach as well as the opportunity to further study and explore this topic.

2. TO DB OR NOT TO DB?

This section provides more concrete motivation of why using DBMS can be of significant importance in new scenarios like scientific databases, as well as of the fact that drastic changes are needed, and ends by discussing our vision and research challenges.

As a starting point we perform a study of how using a DBMS compares against using Unix tools for query processing. We compare the popular and powerful Awk scripting language with a state of the art open-source column-store DBMS, MonetDB. We use a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM and two 500 GB 7200 rpm SATA hard disks configured as software-RAID-0. The operating system is Fedora 12.

The set-up is as follows. The data set consists of a four-attribute table, which has as values unique integers randomly distributed in the columns. The queries are always 10% selective, and follow the template below (Q1):

```
    select sum(a1),min(a4),max(a3),avg(a2)
Q1  from R
    where a1>v1 and a1<v2 and a2>v3 and a2<v4
```

Figure 1 shows the results for various different input sizes and two kinds of costs: the loading cost and the query processing cost. For Awk there is no loading cost, while for the DBMS, the complete loading cost has to be incurred before we can process any data. For the query processing performance of the DBMS, we report both hot and cold runs, as well as evaluation over an adaptive indexing scheme. In the next two subsections, we study these results from two perspectives; In the first part, we discuss clear disadvantages for the DBMS, while in the second part we discuss clear DBMS advantages. The third subsection then introduces our vision for a hybrid system.

2.1 Why Not Databases

Let us first discuss the disadvantages of using a DBMS.

Loading Overhead. With the term "loading" we refer to the procedure of copying the data from the flat files into the DBMS. Our flat files are in CSV format. To process even a single query in a DBMS we first have to incur the *complete* loading cost. Thus, the first query response time can be seen as the cost to load the data plus the cost to run the actual query. Figure 1 shows that the loading cost of the DBMS represents *a significant bottleneck* when it comes to large data sizes. In other words, loading the data and evaluating a query with the DBMS is much slower than simply firing an Awk script. If we add the loading cost and the query processing cost of the DBMS for the case of 1 Billion tuples, it is 800 seconds higher than that of running an Awk script. And of course, here we ignore the expert knowledge and time required to set-up the DBMS (e.g., with the proper physical design).

In addition, while the query processing performance of Awk scales perfectly with input sizes, this is far from true for the loading cost of the DBMS. Recall that the loading cost is to be considered as part of the first query in a DBMS. What actually happens is that for the smaller sizes everything fits quite comfortably in memory and is never written to disk during the experiment. For the input of the 1 Billion tuples table, however, the system reaches the memory limits and needs to write the table back to disk; exactly then, we pay the significant cost of the loading procedure.

Not a "Quick Look" to the Data. This experiment represents the *ideal* scenario for a DBMS, i.e., the query is interested in all 4 attributes of the table. This way, even for the case of 1 Billion tuples where loading is a significant

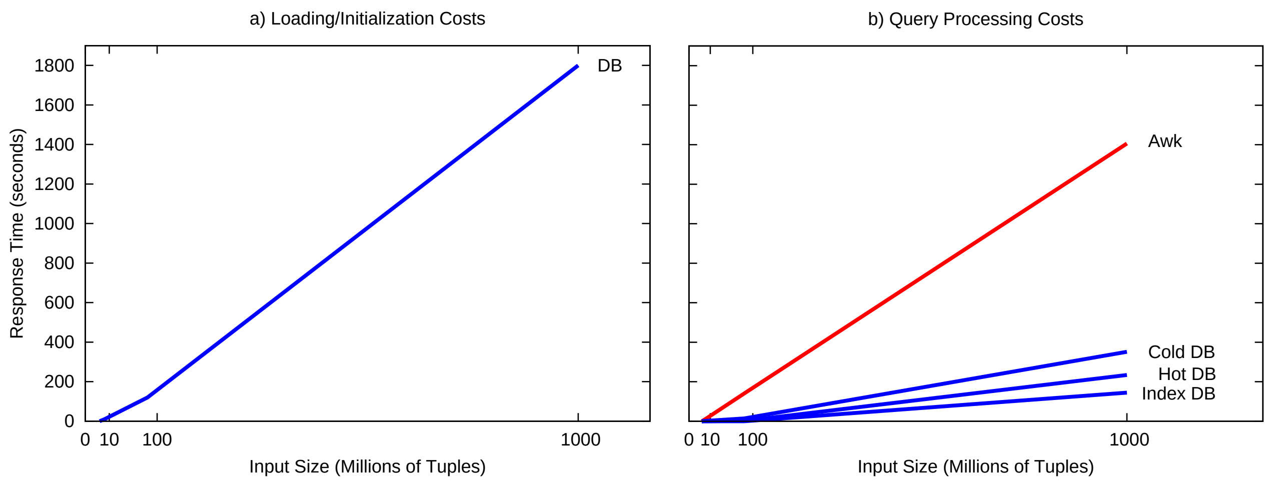

Figure 1: DB Vs. Unix tools

cost, the DBMS did not do any extra work, i.e., it did not load any unnecessary columns.

In scenarios like scientific databases though, it is typical to have hundreds or even thousands of columns, while a query might be interested in only a few of those. A DBMS has to first load all columns before it can process any queries. It has to *own and replicate* the complete data set. On the contrary, a scripting tool like Awk can provide a quick and "painless" gateway to the data. We can explicitly identify the data parts we are interested in, while zero preparation or pre-processing is required. We do not need to replicate the data and we can actually edit the data with a text editor directly at any time and fire a query again.

The above observations make scripting languages like Awk a much more flexible and useful tool than a DBMS for inspecting and quickly initiating explorations over large data sets.

2.2 Why Databases

The previous section argues that Awk is great for a quick glimpse of the data. The DBMS is not. For repeated queries over the same data, however, Awk lacks the smarts to improve performance; this is where the DBMS shines! Here, we discuss these DBMS benefits.

Performance Gains After Loading. Let us ignore the loading costs and concentrate purely on the query processing costs in Figure 1. For all sizes, we see a consistent pattern with the DBMS being significantly faster than the pure performance the optimized Awk script can provide. No matter if we look at cold or hot queries the DBMS is much stronger. The indexed DB curve indicates an even better performance using an optimized physical design, where storage is adapted to the queries via index creation (in this case selections and tuple reconstruction are significantly improved using database cracking [12]). Especially as the data grows, the DBMS is one order of magnitude faster. By having the data loaded and stored in a proper format, the DBMS does not have to go through an expensive tokenization and parsing procedure again and again; it only pays this cost during loading. Consequently, even the non-indexed DBMS runs can materialize significant benefits over Awk that always has to go through the flat files.

Exploratory Behavior Benefits. Advanced data management scenarios like the analysis of scientific data is far from an one query task. It typically involves a lengthy sequence of queries which dynamically adapts based on how the scientist interprets the data, continuously zooming in and out of data areas representing an exploratory behavior. A scripting tool has a constant performance that cannot improve over time. A DBMS, on the other hand, has an one-time up-front cost and from there on performance is very good making it a great fit for when we know we are going to query again and again the same data parts. In addition, indices, optimizers and advanced operator implementations guarantee near-optimal usage and exploitation of hardware and workload properties. A DBMS can be tuned to properly scale to hundreds of nodes or CPUs, it can be tuned to adapt to skew of data or queries, etc. On the other hand, scripts typically are single threaded processes without the notion of optimization and adaptation.

Fast Evaluation of Complex Queries. Experiments with more complex queries show even bigger benefits. For example a join query with a few aggregations on two 10^8 tuples tables (1 to 1 join) takes 387 seconds on a hash join implementation in Awk, while it takes 247 seconds if we sort the data (using the Unix sort tool) and then implement a merge join in Awk (a 100 lines script). On the contrary, a cold DB run takes 39 seconds while a hot one only 5!

Declarative SQL Interface. Expressing queries in the declarative SQL language is a major benefit of a DBMS for flexibility and query reuse/editing. On the contrary, with Awk one has to be an expert in scripts. In addition, workload knowledge is needed for optimal performance. Our scripts "match" the techniques used in an optimized DB plan, i.e., push down selections, perform the most selective filtering first, etc. Even though using Awk in everyday scripting is an invaluable tool, the overhead of writing complex scripts to match SQL expressiveness should not be underestimated. Our experience shows that, even for an expert, it requires several hours to produce efficient scripts. A simple 1-2 line SQL query needs several tenths or hundreds of lines in a scripting language. Thus, the even harder part becomes that of maintenance and reuse. In our interaction with scientists in EPFL, several of them admit that they have numerous scripts that they either never attempt to edit or they do not remember any more what they do.

We repeat the above experiments using also Perl instead of Awk. This was two times slower than the Awk scripts. Another heavily used tool is Matlab. It specializes in scenarios where heavy computation is needed. We did not try it out in our experiments but nevertheless the argumentation remains the same as with all scripting and low level languages. Similarly, one may actually write pure C code which is expected to be faster for targeted queries than a generic DBMS.

None of the methods above provides the flexibility and scalability of a DBMS and of course they require expert technical knowledge as well as a good understanding of the data. DBMS were built based upon this motivation; *generic, abstract and declarative* usage of an information management system that can actually perform very well. The users need to care only about what they want to obtain from the system and not about how to obtain it.

2.3 Vision & Challenges

The previous sections showed both major advantages and disadvantages for using a DBMS. Here, we discuss our vision towards a hybrid system that blends the best of scripting tools and DBMSs.

2.3.1 The Vision

The ultimate goal is a system that avoids all the initialization steps and hassles of a DBMS, but at the same time it maintains and even enhances the potential performance gains.

All you need to do to use it, is point to your data and you can start querying immediately with SQL queries.

Without any external administration or control and without any preparation steps, the system is immediately ready to answer queries. It selectively brings data from the flat files into the database during query processing. It adaptively stores these data into the proper format, adjusting its access methods at the same time, all driven by the current hot workload needs. Column-store, row-store and hybrid storage and execution patterns all co-exist even for the same or overlapping parts of the data set.

2.3.2 Challenges

The challenge is to make all this continuous and ever evolving process as transparent as possible to the user via lightweight adaptation actions and rapid response times. Conceptually we strive to blend the immediate and interactive feeling and simplicity of using Unix-like tools to explore data with the flexibility, performance and scalability of using a DBMS. Some of the main research questions we have to answer are the following.

1. When and how to load which parts of the data?

2. How to store each data part we load?

3. How to access each data part?

The first one represents a completely new problem. Up to now in order to use a DBMS we need to completely load the data. There is no notion of partial, incremental and adaptive loading.

For questions 2 and 3 the database community has done extensive research to find the optimal storage and access patterns for *specific* workloads and scenarios. Nevertheless, there is no notion of an adaptive, ever changing, dynamic storage *and execution* kernel.

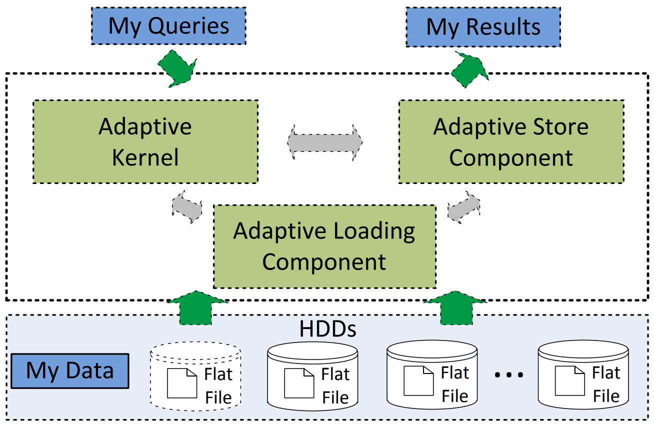

Figure 2: System Architecture

2.3.3 Basic Components

The above goals open a broad research landscape, which touches every corner of core database systems design. Loading, storage and execution, they all need to adopt a self-organizing nature, capable of adapting to the workload, requiring zero or minimum human effort. The key to our research path is that:

Queries become the first class citizen that define loading, storage and execution patterns and strategies.

The components of the envisioned architecture as well as the ways that these components interact can be seen in Figure 2.

An *adaptive loading* component will be responsible to always make sure that just enough data is loaded but also that we can easily access the rest of the data if needed.

An *adaptive store* component will make sure that data is stored in ways that fit the workload.

In the same philosophy, an *adaptive kernel* will at any time contain multiple different execution strategies to best suit the observed workload.

In the rest of this paper, in Sections 3 and 4, we provide an initial study and an implementation prototype of adaptive loading. This is the first critical component needed to realize this vision. In addition, it probably represents the most provocative of all challenges given that database systems always like to fully *control and manage* the data in their own format and environment as opposed to relying on flat files. Section 5 sketches the research landscape. It discusses in more detail the challenges and opportunities associated with the adaptive store and the adaptive kernel concepts. We also provide an extensive discussion of how several core database design problems need to be rethought in this vision, e.g., issues that have to do with updates, concurrency control and robustness.

3. ADAPTIVE PARTIAL LOADING

In this section, we study adaptive loading in more detail, providing a series of techniques as well as an implementation in a complete system.

3.1 Adaptive Loading Techniques

The goal for adaptive loading is to avoid the expensive and resource consuming procedure of loading the complete data set when this is not necessary. The direction is then to only partially load data and the questions to answer are: *when* we load, *how much* we load every time and of course *how* we load.

For this initial study we will for simplicity assume a columnar layout to focus purely on the loading part. This way, here when we refer to data loading, this happens at the granularity of columns (or parts of it) resulting purely in an array-based storage form underneath. Row-store formats and hybrids, they all pose slightly different challenges waiving away some of the problems a column format poses but at the same time adding new ones. The analysis below demonstrates the potential of adaptive loading and indicates potential benefits from further detailed studies.

3.1.1 When

Assuming dynamic and evolving scenarios without upfront clear knowledge of what to expect, the decision is to load dynamically and partially *during query processing.* This way, we achieve the zero cost initialization property as all loading responsibility is transferred to queries, just as it is with scripting tools.

3.1.2 How Much

Again, assuming no workload knowledge, the decision is to do the minimum possible investment at every time. This way, for each query we process, we make sure we have all necessary data in order to correctly and completely answer it. In other words, incoming queries not only trigger loading but also define how much we load based purely on their needs.

One direction here is to load complete columns at a time, having only the hot columns loaded. When a query comes, make sure that all needed columns are there and if any are missing, then go back to the flat file to load the needed part.

A second direction is to only partially load columns in order to reduce the loading overhead even more. This can be thought of as *pushing selections down into the loading phase.* Loading only qualifying data ensures a minimum possible per query overhead and a minimum possible storage footprint. This is ideal for exploratory scenarios where the user "walks" through the data space, periodically zooming in and out of specific data areas. However, an extra cost and complexity is involved in needing to maintain a table of contents so that we know what portions of a column are loaded. Naive strategies might have to go back to the flat file too often.

3.1.3 How

We can identify two topics here. First, how do we fit these techniques in the software stack of a DBMS. Second, how do we actually access the flat files in an efficient way.

We design new adaptive loading operators which are plugged into query plans and are responsible to bring the missing data. These operators mimic the pure loading procedure of a DBMS. The difference is that they can *selectively* bring data from a file. Such an operator can load, on-the-fly, any combination of columns from a flat file. We also experimented with operators that load only one column at a time. This is simpler to design but it turns out to be much more expensive due to the need to touch the flat file multiple times within a single query plan. For each column, we also keep additional metadata information that indicates whether this column is loaded or not, and if it is loaded then we maintain the information of which parts are already loaded and where and how they are stored. A tree structure that organizes the data parts of each column based on values is sufficient, e.g., an AVL-tree or a B-tree.

The general idea is that after all optimization of the original query plan is finished, a new optimizer module/rule takes over to rewrite the optimized plan into a query plan that properly contains the new loading operators. For each column that is marked as not fully loaded we potentially need to act. For each table referenced in the plan, the optimizer will add one adaptive load operator to bring in one go all missing columns or parts of them based purely on which columns or parts we miss for the current query. This way, the system can handle in the same query plan both kinds of tables (already loaded or not), as well as combinations of columns and tables which have different portions loaded at any given time.

In our current implementation and query plans, these operators are plugged in as close as possible to the operators that need the relevant data. Although this is a very reasonable first approach, one can think of optimization scenarios where the loading operators are plugged in dynamically while the query is running, when the system realizes that it has the resources to do some of the extra I/O and processing necessary. In addition, here, for simplicity we assume that the original DBMS optimizer rules are optimal for the new kernel as well. Naturally, this may not be true (at least not always) opening a research line that introduces specific optimizer rules and plans for such adaptive systems. Taking into account the expected costs and delays of adaptive loading operators, with respect to the actual work that they need to do for the current query, might lead to different decisions early in the optimizer pipeline.

3.2 Experimental Results

Here we provide early results of our implementation in an open-source column-store, MonetDB. We test our implementation against the normal MonetDB system and against the MySQL CSV engine. The latter is a recent engine option of MySQL which allows to directly query flat CSV files. It provides the flexibility of querying a flat file with SQL but it does not provide the DBMS benefits as it resembles a Unix-tool like Awk in that it needs to read the data again and again for every new query, i.e., it does not load the data in any way, optimize the layout, etc.

We implemented new adaptive loading operators and query plans in MonetDB. These operators can be plugged in query plans to provide on-the-fly loading of only the necessary data. From a high level point of view the operators work as follows. The flat file is split into multiple portions horizontally. Then, multiple threads take over to tokenize each portion. Tokenization is done in two steps for each file portion. First, we identify where each row begins. Then, within each row we find out where the relevant columns are. While

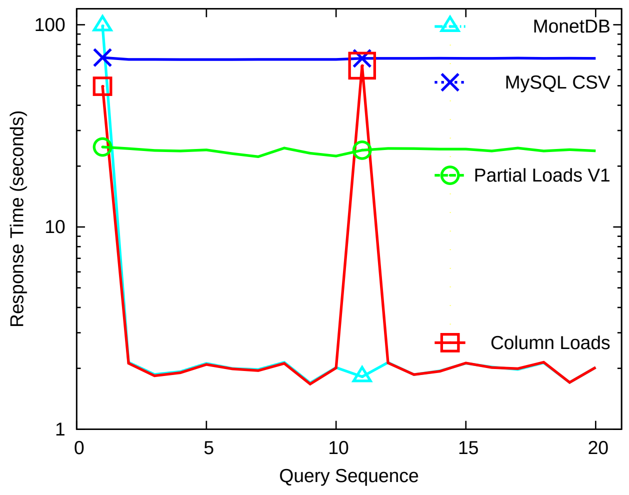

Figure 3: Alternative Loading Operators

tokenizing a given row to identify the needed columns, once all required columns are found the tokenization for this row can stop., i.e., there is no need to tokenize any columns not needed for this query. In addition, if the *where* clause is pushed into the loading operator, then once a needed attribute is tokenized, it is parsed and the relevant predicate, if any, is applied on-the-fly. This allows us to abandon the tokenization of a row as soon as a predicate fails to qualify for this row. After all tokenization is done, then multiple threads take over to parse the values and put the proper values in the proper columns and in the proper order. The process continues until all horizontal portions are read.

Figure 3 shows results on a 10^8 tuples table. It contains 4 attributes of unique integers and we use Q2 queries as follows.

```
   select sum(a1),avg(a2)
Q2 from R
   where a1>v1 and a1<v2 and a2>v3 and a2<v4
```

Each query uses two attributes and is 10% selective. Here we first run 10 random queries that use the first two attributes of the file and then we run another 10 that use the last two attributes.

The MonetDB curve represents the normal database behavior where everything is fully loaded up-front. Here, the complete loading cost is attached to the very first query, representing a significant delay. Every query after that is fast as it exploits the already loaded data (no indices are created here). The CSV engine of MySQL demonstrates a stable performance. Having to read and analyze the complete flat file for every query brings a constant response time throughout the query sequence. The Column Loads curve represents our new query plans in MonetDB that on-the-fly load the needed columns missing from the current storage. The cost of the first query is roughly half the cost of the normal database query representing a significant gain. The next 9 queries enjoy very good performance similar to that of plain MonetDB. These queries also need the same attributes as the first one, so all needed data is already there and no extra actions are needed, leading to similar performance as that of MonetDB. Query 11 though needs a completely different set of data and thus it on-the-fly needs to load the missing columns. However, again this is much faster than a complete load and all queries after that can exploit this to gain good performance. Overall, both the MonetDB curve and the Column Loads curve spent similar amount of time in loading. However, the Column Loads curve spends this cost only when it is needed, allowing for faster initialization of data exploration. The cost is spent only when and if necessary, i.e., if the workload never demanded some of the columns of the file, those would never be loaded.

Finally, the Partial Loads curve represents query plans with loading operators that can perform filtering on-the-fly. Only the values that qualify the where clause are loaded, allowing the loading operators to shed extra cost by minimizing the actions on non-qualifying data. Complete tuples in the flat file can be ignored as soon as one of the predicates in the where clause fails for a given tuple which allows the adaptive operator to immediately stop any further parsing and loading actions in this row. This kind of partial adaptive loading has the side-effect of creating intermediate results that are identical to what a selection operator over the complete column would create. This way, the query plan can then continue from this point on, avoiding all the where clause operators.

For this specific experiment, Partial Loads throws away the data immediately after every query. Thus, this represents the potential benefits of the most lightweight direction of never “really loading” anything, never paying the I/O cost to write the data back to disk and always reading just enough from the file by exploiting early tuple elimination while parsing. However, given that we have to go back to the file for every query, as MySQL does, performance remains quite stable with no room for improvement. Multiple variations of the above are possible, shaping up an optimization problem. In the next section, we present a variation where Partial Loads does not throw data away and future queries can actually reuse them if they overlap.

4. DYNAMIC FLAT FILE ADAPTATION

The previous section showed some very promising results. Loading can be done incrementally and it can be added in the software stack of a modern DBMS. Given though the inherent costs of reading and analyzing flat files, these results indicate that a significant cost is involved every time we need to read from a file due to missing data. Here, we study this problem and sketch a solution that goes even more beyond the traditional loading schemes.

4.1 Split Files On-the-fly

4.1.1 I/O Overhead

Going back over and over to the full file has the inherit cost of requiring to bring the complete file from disk, tokenize and parse a part of it over and over again. When the complete file needs to be loaded, it is justified to pay this cost. In our case, though, we are incrementally loading the data and we have to go back to the file at multiple occasions; flat files are organized in rows but the queries ask for specific columns. Thus, this is exactly the same problem as in the typical row-/column-store environment where a column-store has the advantage of selectively reading only the necessary columns for a query, while a row-store needs to read everything no matter what the query needs are.

4.1.2 Tokenization Overhead

A second problem with a monolithic flat file is that every time we need to load a different attribute or part of it, we need to tokenize *all* attributes that appear before our target in the file. For example, if we want to locate the 5th attribute in each row of the file, we need to tokenize the previous 4 so that we know when we reach the 5th one, even though these attributes are not relevant for the current query. If for a future query we need to load part of attribute 6, then again we need to tokenize all previous 5. Maintaining information on where each attribute starts is prohibitive as this is different for each row; assuming fixed length attributes is unrealistic in general. Even if we had such knowledge, we still have the inherit cost of needing to read the complete raw file from disk even if only a few of the values are needed.

4.1.3 Splitting Flat Files

The direction to follow here is to try to amortize the loading cost over the sequence of queries, making sure that we do not perform the same actions over and over again. To achieve this, we have to eliminate the need to repeatedly read the complete file. At the same time, we want to avoid tokenizing the same part multiple times. Both of these goals can be achieved if we incrementally and adaptively *split* the file during the loading phase such as future loading steps can locate the needed data much easier.

For example, say we need to load column A. With the techniques of the previous section, we have to go through every row of the file and tokenize it until we locate column A. Then, we parse and load this value and ignore the rest of the row. Thus, for every row of the file, we have paid the cost of tokenizing every attribute before A, but we have not tokenized any attribute after A.

4.1.4 Dynamic File Partitioning

To exploit our current efforts in future queries, one direction is to on-the-fly create a few supporting structures as a side-effect of loading. For example, here we can create one new flat file for each attribute we tokenized and one flat file for all attributes we did not tokenize. This way, when in future queries we need to load any attribute which is already tokenized, we can simply read the respective input file containing only the needed values. The befit is twofold. First, we completely avoid the overhead of reading the rest of the input data. Second, loading the column becomes much simpler since the file contains only the needed column and thus tokenization involves only separating the values by recognizing the end of each row. Even if we need an attribute from the non-tokenized ones, again we gain by reading and analyzing only part of the flat file and thus progressively making the loading cost less and less as we gain more knowledge.

4.1.5 Learning

Conceptually this can be thought of as "file cracking", i.e., similar to database cracking [12] that dynamically reorganizes columns inside a DBMS based on queries, file cracking dynamically reorganizes flat files to fit the workload needs and ease future requests. The concept of cracking is to learn as much as possible when touching data and with lightweight actions reflect this knowledge. The same mentality can be applied here with several research directions to study towards on-the-fly maintaining a more sophisticated table of contents over the flat files. Every time we touch a file, we learn a bit more about its structure, e.g., the physical position of certain rows and attributes. Solutions that provide a more active reorganization of the flat files are also possible in order to introduce even more knowledge. Identifying and exploiting this knowledge in the future can bring significant benefits.

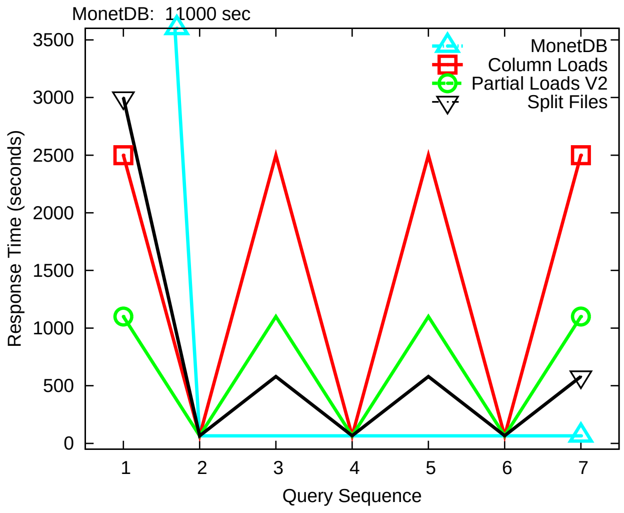

Figure 4: Adaptive loading with file reorganization

4.2 Experimental Results

We implemented the split file functionality in our MonetDB implementation of adaptive loading. During tokenization, the already seen columns which do not qualify for the current query are not ignored as before. Instead, pointers to the values of each column are collected into arrays and once all tokenization is finished, they are written in one go in one separate file per column. The non tokenized columns are written in a single separate file. The system then has to update its table of contents regarding where subsequent queries can find each column.

Figure 4 shows the results. We use a 12 column table of 1 billion tuples and Q2 queries. Every 2 queries we use 2 different attributes of the table until all attributes have been used, running 12 queries in total. In order to emphasize the best and the worst case for each technique, the second query in each run is simply a rerun of the first, i.e., we run 6 different queries, and each query runs twice. The y-axis in Figure 4 is trimmed at $3.5 * 10^3$ seconds while the first query of MonetDB reaches $11 * 10^3$ seconds. In addition, for better presentation the x-axis shows the first 7 queries as the rest of the sequence maintains the same performance patterns.

The performance of MonetDB and Column Loads is similar to our previous experiment; MonetDB loads the data completely with the first query, while Column Loads loads incrementally whenever missing columns are needed. With Column Loads we load only the necessary columns every time essentially amortizing the loading cost across the workload. Query 1 loads fully 2 of the columns. Then Query 2, needs the same columns and thus it can provide a performance similar to MonetDB as no extra actions are needed; not all data is loaded but all necessary data is there. The Partial Loads V2 manages to produce smaller peaks by selectively loading only the necessary values. Unlike the experiment in Section 3, here Partial Loads maintains the data between queries providing incremental loading.

The effect of the Split Files technique is that it can produce even smaller peaks. In this case, in order to emphasize the best and worst behavior, the very first query asks for the two attributes that appear last in the flat file. This way, it is the first query that does all the heavy work of splitting the *complete* file and thus demonstrates the worst first query case. Even so, the start-up cost is roughly 4 times smaller than that of MonetDB. In the general case, the split file efforts will be amortized among many queries and thus the performance of the first query will be even more attractive. If we look at what happens after the first query, then Split Files achieves an ultimate performance similar to that of MonetDB and thus providing all performance a modern DB can promise. In addition, compared to Partial Loads and Column Loads it reduces the cost of dynamically going back to the flat file, i.e., it is 2 times faster than Partial Loads and 5 times faster than Column Loads. The gain comes purely by the need for less I/O and parsing effort. Every time something is missing, Column Loads and Partial Loads have to go back to the complete raw file, while Split Files can just read the individual files containing the minimal subset needed for the query. This time, this represents the best possible behavior given that the first query has already done all the file splitting.

4.2.1 Summary and Research Questions

The analysis above demonstrates a clear potential. It also raises several research questions, e.g., how much of the flat file should we bring inside the DBMS? By bringing more data than a query needs, we penalize single queries with actions that might never prove useful. By bringing exactly what a query needs, we have higher chances of needing to go again back to the flat file, penalizing queries that represent a shift of the workload. Splitting the file dynamically helps but it potentially doubles the needed storage budget and loses (or makes more complex) the ability to directly edit the flat files via any text editor and immediately fire an SQL query. Quantifying and modeling the various actions is also needed to take educated actions.

5. RESEARCH LANDSCAPE

In this paper, we set a challenging goal towards fully autonomous systems that consider flat files as an integral part of their structure. The goal is to achieve a zero initialization overhead; just point to the data and start firing SQL queries, progressively achieving similar performance with that of a modern DBMS.

The initial designs and experimentation shown in the previous section verify that this is indeed a feasible direction. We clearly show that what is considered as a given fact in modern database systems regarding complete and expensive up-front data loading can be reconsidered.

The task of a complete study of this vision expands over several core database topics and requires an in depth analysis. In this section, we sketch the most important of these topics towards our vision and we provide initial ideas and discussion for the adaptive storage module, the adaptive kernel module as well as for updates, concurrency control, robust performance, etc.

5.1 Adaptive Store

The storage layout is of critical importance. In our vision, storage is created on-the-fly as data is incrementally brought from the flat files into the system. In fact, the storage layer consists of two parts: (a) the flat data files and (b) the data that the engine creates to fit the workload, the *Adaptive Store*.

5.1.1 Continuous Adaptation

The adaptive store is an ever changing layer that continuously adapts to the workload. The key point here is that all this happens on-line when we actually have information that we can exploit towards making these decisions. The queries themselves provide this information. This way, we can continuously create the "optimal" representation for the query at hand.

In the adaptive store the notions of base data, index, materialized view and projection are blurred. The adaptive store may contain data in *any format*, i.e., row-store, column-store, as well as PAX and its variations. For example, while loading data dynamically and partially, instead of merely creating columns to store the data, the system can choose the optimal format for the current query that triggered this loading action.

Multi-format Data. In the same spirit as with the above observations, it is not necessary that all data for a given table follows the same format. On the contrary, different parts of a table may follow completely different formats. Queries dictate the format of the data parts they need and different parts of the same table may be of interest to different queries. By letting independent queries make independent decisions on how to store the missing relevant data and how to exploit existing already loaded relevant data, the system self-organizes to match the workload.

Multi-format Copies. The same part or overlapping data parts may be replicated multiple times in multiple different formats in order to service different kinds of queries.

Multi-file Data. A single data instance in the adaptive store may contain snippets from multiple raw data files, effectively providing partial denormalization as needed by the queries.

Data Padding. Data padding is a powerful tool to fully exploit modern hardware. For example, [15] shows that data padding techniques can successfully be integrated with execution strategies that are aware of the padding and through bitwise operations provide efficient database query processing. A system that can exploit proper padding in an adaptive way and combine this with the rest of the data placement and execution decisions discussed above, leads to an optimal performance, adapting to its hardware potential.

5.1.2 Strategies

There are two extreme strategies one may implement regarding how to manage the adaptive store. In the first one, we target for simplicity, i.e., for every incoming query that is not fully covered by the data which is already loaded, we blindly treat this query as if all the tuples it needs are missing. This means that we will fetch everything from the flat files, including tuples that potentially are already loaded. Then, we create a completely new view to store the newly loaded data. In the second extreme, we always try to minimize the footprint of the adaptive store, i.e., for every incoming query we identify the missing tuples and we load only those. Then, we have to update the proper structures in the adaptive store to reflect the new information as well as combine the existing relevant tuples such as to answer the current query.

Naturally, the second approach is more complex than the first but it gives more flexibility to exploit the adaptive store and cover a bigger percentage of the active workload. The research challenges here are in designing efficient techniques to implement strategies like the one described above as well as study the field between those two extremes.

5.1.3 *Life-time*

Multiple interesting scenarios may be studied regarding how long we need to keep data alive. Data parts loaded via adaptive loading and stored in any format may be thrown away at any time. The only cost is that of having to reload this data part if it is needed again in the future.

For example, one approach can be that the adaptive store is purely memory resident. In this case, loaded data is never written to disk and stay alive only as long as there is enough memory. Once a more "useful" piece of data is needed, an old one is thrown away. Alternative approaches may exploit flash disks as an intermediate layer such as to minimize the effort when data needs to be reloaded.

Designing efficient strategies and algorithms on how to maintain data is of significant importance as it can affect drastically run time performance.

5.2 Adaptive Execution

Proper data placement adapted to query processing can give significant boost in performance by optimizing I/O and cache performance. However, in a system that supports multiple data layout formats, we also need multiple processing strategies to fully exploit the potential of the different layouts.

For example, a pure column-store uses a drastically different execution strategy than a pure row-store in order to benefit even more from the columnar layout. This translates to different operators and different data flows, leading to very different implementations of the DBMS kernel as well as the optimizer. Pure column-store operators, operate on one or at most two columns at a time. Their advantage is simple code, data locality and a single function call per operator. The disadvantage is that they need to materialize intermediate results which may prove fairly expensive when for example we need to operate on multiple columns of a single table. Row-store operators, on the other hand, operate in a volcano style passing one tuple at a time from one operator to the next. No materialization is needed but numerous function calls are required.

5.2.1 *Adaptive Kernel*

In the same philosophy as with the adaptive store envisioned above, here we argue towards an *adaptive kernel* where at any given time multiple different execution strategies are possible to better fit the workload.

The kernel may contain any kind of operator implementations that better fit the data in the adaptive store based on the query patterns. And should be able to create any kind of query plan to better exploit those operators.

5.2.2 *Hybrid Operators*

Other than using pure row-store or column-store operators, there is also a clear space for hybrid operators. For example, when we need to compute an aggregation over three attributes, a new operator that in one go computes the total aggregation would provide the best result, i.e., operating in a column-store like fashion but with a row-store like input.

Combined with optimal hybrid storage, hybrid operators are expected to have a significant impact on system behavior as it essentially means that we always get the optimal storage and access patterns.

5.3 Auto-tuning

The adaptive store and the adaptive kernel envisioned in the previous sections aim in always maintaining the best possible layout and execution mechanisms as the workload evolves. The adaptive loading procedures keep feeding the store and the kernel with just enough data as needed.

Several challenges arise to satisfy all these requirements. Probably the most challenging of all is the question of how the system reaches a good set-up as well how it adapts when the requirements change again. Up-front materialization of all possible storage patterns is not feasible given the numerous combinations leading to storage and maintenance problems. Similarly, up-front implementation of all possible operators/access patterns is not possible given the numerous combinations of inputs and operations to be considered. This way, a dynamic mechanism is needed to on-the-fly create not only the appropriate data layout but also the appropriate operators and plans to match the current workload.

The main challenge is to define and develop the adaptation mechanisms and steps that transform the raw data and the initially "ignorant" kernel into the best fit system for a given workload. Other than the technical challenges, for this we first need to answer the following (high level) questions:

1. What is the best execution pattern for a query set?
2. What is the best storage layout for a query set?
3. How and when we actually adapt?

5.3.1 *Identifying Optimal Strategies*

Questions 1 and 2 require an extensive benchmarking of all possible hybrid storage and execution patterns to understand and model the performance. Even though extensive and important research was introduced in recent years, still one cannot argue with certainty which storage and execution strategy is the best for a given scenario especially when hybrids strategies and execution are thrown in the equation. This is an open research question.

5.3.2 *Adaptation Triggers*

Question 3 goes deep into the new architecture characteristics and of course hide numerous more fine-grained questions. As we discussed earlier, in order to assist the adaptive nature, the initial choice is to trigger adaptation based purely on the query needs. Changes in the storage or execution layer become a side-effect of query processing. Nevertheless, several issues naturally arise regarding what can trigger and guide adaptation. One could consider multiple queries at a time, system and hardware parameters, etc.

5.3.3 *Re-organization*

The adaptive store and the adaptive kernel are ever evolving structures. They continuously change their shape to fit the workload but also to adapt to restrictions, e.g., storage restrictions. Data in the adaptive store may be reformatted on-the-fly if the workload indicates towards a more fine-tuned storage layout. Similarly, the kernel may alter its execution strategies to follow such changes.

Furthermore, existing data and operators may be thrown away at any time to make room for new ones based on the workload. This may be done, either to deal with storage restrictions, to ease updates or simply to lead to a more lightweight and thus better performing system.

5.4 Updates and Concurrency Control

Updates and concurrency control is one of the hardest topics in database design. In our proposed vision these topics become slightly more complex due to the continuous data reorganization. There is though a plethora of interesting ways to resolve these issues.

Regarding updates, the issue is how to update an already loaded table R. There are two reasons why we would like to update such a table. The first one is that the flat file has changed, e.g., because the user updated it via editing with a text editor. The second one is that we might want to extend R with more data. Recall that in our design loaded tables are partial views of the actual data that belong in a table. The data materialized reflect the workload needs and it is based on design choices. For example, we have seen techniques in this paper, where in a column-store architecture we do full or partial loads. When we do full loads, this means that the very first query that requests a column A, will completely load column A, while with partial loads it will only load the portion of A needed. The second choice means that future queries needing A and not covered by the loaded portions, will need to update the existing structures. The second scenario brings concurrency control issues as well. Multiple queries might be asking for the same column at the same time, meaning that these queries have to touch and update the same loaded table with data brought from the flat file.

One easy solution to all the update and concurrency control scenarios is to treat each request independently and each table as a completely auxiliary data structure that we are not afraid to lose. For example, each incoming query that is not fully covered with whatever is already loaded in the database will be treated as if it is the first query for the data needed. This way, it will create its own table portions but without incurring any conflicts on the way. Similarly, every time a flat file is updated, we can simply drop all relevant tables that have been created with data from this file.

The advantage of the above approach is simplicity. The research challenge here is to investigate differential methods and proper locking strategies for updates such as to avoid extensive data replication, share as much work as possible between concurrent queries and to avoid extensive trips back to the flat files.

5.5 Robust Performance

Any adaptive method may suffer from a non robust performance. For example, the worst case scenario in a system with adaptive loading is one where the data parts loaded are never used. Say, we have a memory limit of X bytes. Queries arrive and start creating more and more tables and feed them with data from the flat files. Say that no query is covered by any existing table and thus all queries have to go back to the flat files. Then, we reach the storage limit and we start dropping tables. Thus, all the effort of incremental loading is wasted.

Another similar scenario is the case of incrementally loading a column with partial loads where each query brings from the flat file only the portions that it needs. Individually each query is much faster than one with full loads as we saw in our experiments. However, in the worst case scenario we will have to go back to the flat files as many times as the tuples in a column. Say for a column of N tuples we receive N queries one after the other where each query fetches only one of the N tuples. In this case, doing a full load with the first query is a much better approach as it avoids the extensive I/O.

Thus, the challenge, for providing a robust performance relates to a continuous process to monitor the system performance and the workload trends such as we can continuously adjust critical decisions that may significantly affect performance, i.e., how much data to fetch from the flat files with every query, whether to replicate tables or use a single instance of each one, whether split the flat files or not, etc.

5.6 Schema Detection

Another important research topic is automatic schema discovery. When the user links a collection of flat files to the database, a schema should be defined. Ideally, this should be done without any input from the user. Following a simple strategy, each flat file can be mapped to a table. In the tokenization phase, the columns of a given row are identified, each column becomes an attribute of the table and we examine each attribute to figure out its proper data type. This task is performed only once during the first query execution.

Nevertheless, all schemas are not equally good; many times integrity constraints and functional dependencies are important. Advance techniques such as database normalization, data de-duplication and data cleaning can be considered and we cannot expect that the structure of the flat files reflects a good schema for the query workload indented.

6. RELATED WORK

There have been several important milestones in the research literature that brought us to this research line. In this section, we briefly discuss related work that inspired the vision of this paper. There has been related work in multiple aspects of database research, mainly related to external tables functionality, self-organization and hybrid storage layouts.

6.1 External Tables

Flat files were always considered "outside" the DBMS engine. The general guideline is that a DBMS has to load the data completely before it can do anything fancy with it. Recently, there have been a number of efforts both in industry and in academia to partially attack this problem.

FlatSQL [16] claims the usefulness of using SQL to express queries over flat files. The main motivation is that one can use SQL, i.e., a declarative language as opposed to a scripting one to interact with the system. Under the covers, FlatSQL translates SQL to Awk scripts so it can actually query the flat files. This work was mainly motivated by scientific applications and it shows that the motivation for using a DBMS is not only raw performance but also several of the features that make DBMSs friendly to the user.

In [22], the SciDB project presents a collection of essential features needed for modern scientific databases with huge data loads. The ability to query raw data by minimizing the overhead of loading was included as one of these features that modern DBMSs need to support so they can be useful for scientific data analysis.

More recently, several open source and commercial database systems include the functionality of external tables. The idea is that the system can read directly from flat files triggered by an SQL query. Nevertheless, current designs do not support any advanced DBMS functionality. In other words, they provide the same benefits as with FlatSQL but without the need to call Awk. In terms of performance though, this cannot match a normal DBMS as it needs to continuously parse the data.

The vision provided in this paper goes multiple steps further. By allowing not only to access flat files via SQL but also to selectively and dynamically load part of the data and store it in what essentially is an index-like format, we can combine both the ease of using SQL and the superior performance of exploiting a traditional database engine.

6.2 Self-organization

Self-organization has been studied in multiple problems of database research. The main goal of any technique in this front is to reduce or eliminate the need to tune the system which in turn significantly reduces both the user input required and the time needed to reach a highly tuned system. Typically, this means an effort to automatically select and create indices, materialized views, etc.

Auto-tuning Tools. For example, there has been a significant line of work in the area of auto-tuning tools, e.g., [1, 3, 4, 20, 23, 25]. These tools automatically select the proper indices given a representative workload. They can even take into account the available storage budget that we can devote in the auxiliary structures, the cost of updates, etc. Nowadays, these are invaluable tools when setting up a new system. This is typically an off-line approach, i.e., it requires a priori workload knowledge and enough idle time to analyze the representative workload. More recently, there have been efforts in adapting these tools towards a more on-line approach [4, 20]. In this case, the system can start with zero indices, then monitor incoming queries and performance and eventually come up with a set of candidate indices that match the running workload.

Adaptive Indexing. A second line of work is adaptive indexing, i.e., database cracking and adaptive merging [7, 8, 9, 14, 12, 13]. With such adaptive indexing techniques, index selection and index creation happens as a side-effect of query processing. The user does not have to initiate any tuning or provide a representative workload. At any given point in time, an adaptive index is only partially optimized and partially materialized such as to fit the current workload and storage budget. As queries arrive, the index representation adapts at the physical level to fit the workload. For example, the basic cracking techniques can be seen as an incremental quick sort whereas the basic adaptive merging techniques can be seen as an incremental external sort. For example, in the MonetDB implementation of database cracking and adaptive merging in a column-store, each operator (a selection, a join, etc.) physically reorganizes the arrays that represent its input columns using the query predicates as an advice of how the data should be stored.

Blink. Another recent research path is the Blink project from IBM [15, 19, 17]. Recognizing that index selection, exploitation and tuning is a major hurdle, Blink completely removes these steps. The motivation is that the system should be usable with minimum tuning; there are no indices and there is no optimizer. Blink combines several novel techniques to deliver high performance out of the box. Some of the core technologies include denormalization, compression via frequency partitioning and a run-time kernel that can operate directly on compressed data. Frequency partitioning allows to create multiple partitions of the data, collecting similar values from each attribute in the same partition and compressing them with fixed length codes per partition. This way, Blink achieves both high compression and high performance, since at run time the kernel can exploit vectorized query processing. In addition, Blink packs several tuples into CPU registers and operates on blocks of tuples at a time even in compressed form. The end result is high performance with zero or minimal tuning; the user needs to simply load the data.

Our Approach. All research effort described above, relates to ours in that it tries to improve the user experience regarding the tuning effort needed. However, our vision goes a significant step further to introduce the self-organization flavor all the way to the very beginning of the user experience, i.e., *before even loading the data.* All existing efforts help only after the data is completely loaded. We expect that several of these ideas will apply to our vision as well.

6.3 Storage Layout

Trying to create a hybrid store has been the subject of several seminal papers over the years, e.g., [2, 10, 18, 11, 21, 24]. Lately, the issue has received wide recognition by the commercial world as well, with big vendors and new start-ups pushing out hybrid technologies, e.g., Oracle, Greenplum, Vertica and Vectorwise. Previous work was mainly focused on improving the way data is stored, bringing the I/O or cache benefits of a column-store design to a row-store setting but leaving the execution part as is, i.e., using N-ary processing. Other work [26] also demonstrates that significant benefits may come from combining the NSM and DSM execution strategies. Building on top of a pure column-oriented storage layout, it shows that it is beneficial to change between NSM and DSM execution strategies on-the-fly. In fact, attempts for hybrid designs began even earlier when vertical partitioning was the main tool to improve performance with respect to the I/O, e.g., [5].

More recently [6] proposes a declarative language interface to explicitly define storage patterns and data layouts. Such an interface would be of significant importance since it can simultaneously simplify and model the interaction with the lower storage level. It still requires, however, a set-up step which assumes a good grasp of the query and data workload. Thus, this is orthogonal to our vision here.

Contrary to the above literature, we sketch the vision of an adaptive store that automatically feeds itself from flat files, making sure that data fits the workload both in terms of which data parts are loaded and also in terms of which storage and access pattern is being used for each data set.

7. CONCLUSIONS

This paper sets a challenging vision; the user should just give the raw data files as input and the system should be immediately usable. Flat files should not be considered outside the DBMS anymore. Here, we sketch this research path and provide an analysis of how a system can adaptively and dynamically load only parts of the input data, and keep feeding from flat files whenever this is necessary. Hybrid storage and execution techniques will further enable this vision that opens a research line towards systems where each incoming query is treated as a guideline of how to load, how to store and how to access data.

In addition to opening up a new line of research, we expect a significant impact by enabling wide database systems usage across multiple domains. Scientists benefit from faster and (more) complete data analysis thanks to shorter query response times. At the same time, several other application domains can benefit ranging from large corporate set-ups to everyday personal applications. The latter should not be ignored! For example, a person's music or photo collection is typically stored in a file hierarchy, manually organized. With personal data growing in massive numbers and ranging over numerous areas, e.g., music, photo, digital books, movies, agendas, maps, etc., personal data management quickly becomes a horrendous task – but a single user will never go into the trouble of putting his/her data into a DBMS due to the initialization trouble and expert knowledge required (the interface should also change from SQL to natural language but this is an orthogonal issue).

Thus, the path of truly adaptive and autonomous databases applies to a vast range of scenarios and has the potential to have a significantly positive effect on modern life.

8. REFERENCES

[1] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server. In *VLDB*, 2004.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[3] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*, 2005.

[4] N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB*, 2006.

[5] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Software Eng.*, 16(2), 1990.

[6] P. Cudré-Mauroux, E. Wu, and S. Madden. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[7] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPCTC*, 2010.

[8] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *SMDB*, 2010.

[9] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[10] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. VLDB '03.

[11] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

[12] S. Idreos, M. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[13] S. Idreos, M. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, 2007.

[14] S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple-reconstruction in Column-stores. In *SIGMOD*, 2009.

[15] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[16] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS. *Harvard*, 2003.

[17] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1), 2008.

[18] R. Ramamurthy, D. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[19] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.

[20] K. Schnaitter et al. COLT: Continuous On-Line Database Tuning. In *SIGMOD*, 2006.

[21] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *VLDB*, 2004.

[22] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.

[23] G. Valentin et al. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, 2000.

[24] J. Zhou and K. A. Ross. A Multi-Resolution Block Storage Model for Database Design. In *IDEAS*, 2003.

[25] D. C. Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

[26] M. Zukowski, N. Nes, and P. A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, 2008.