Provisional Application for United States Patent

TITLE: UBiquitous indeX matrix computing architecture **(UBX™)**
INVENTOR(S): John Wang, Weimin Zhao, Lawrence Thoman
USPTO Patent Application Number: 24300156-62264695-2690

BACKGROUND

The BIG data customer frequently requires making, analyzing, creating or running:
Predictive non-linear, non-parametric models.
Path-dependent Monte Carlo simulations.
Very complex iterative global optimization.
Quantum covariance matrix calculations.
Query n-dimension stratification or slicing and dicing by variables and filters.
Real-time conditional probability decision with highly skewed fat-tails.
Probabilistic economic value-added outcomes in time series.
Adaptive feedback loop for any incremental or derivative alpha knowledge.

Typical data analysis involves ad hoc query, summation, sorting, and linear table scan. The organization of data and metadata greatly affects the performance and ease of use. When performing ad hoc queries, many existing analytical tools use a combination of indexing of some frequently used fields and linear table scan on non-indexed fields. One of the reasons that only selected fields are indexed is the large size of index metadata itself. It is also time consuming to create indices.
Another commonly faced problem using conventional tools is the flexibility when creating new fields. When new fields need to be added, it usually requires the whole data (table) be recreated. Therefore, conventional tools provide acceptable and sometimes satisfactory data analysis solutions when the data involved remains relatively small. On the hand, when large amount of data is involved, a better approach is needed.

BRIEF SUMMARY OF THE INVENTION

The UBX™ (UBiquitous indeX matrix computing) is designed to deliver speed-of-light analytic performance, when it is needed most. In UBX™ approach to solving this problem, the entire source data sets, from multiple systems of every record and every field, are indexed in parallel across multiple *compute nodes*. A compute node is a separate CPU, each with its own dedicated support subsystems. The indices are locally cached and the analytical operations involving those records are executed locally, in parallel on each of the compute nodes. These operations are synchronized and coordinated by a network of system governors (*SysGovernor*) that also manages the staging of data and the storage of intermediate and final results.
Our invention uses a novel approach to organize the data and metadata to address the shortcomings of the conventional approach.

We store the data and metadata separately. The data is stored in the most natural form: flat file format. All records and fields have fixed width. All fields are indexed and indices are stored individually. The index data contains the sorting order of corresponding fields. This patent is about a mechanism of performing data and mathematical computation in a parallel fashion. It is a parallel adding machine at heart with additional control mechanisms.

There are many examples of data- or computation-intensive applications in scientific, engineering, financial, and other various industrial fields. A good majority of these applications can be decomposed into the simple task of adding up numbers, such as errors of estimated vs. actual at all mesh points in a simulation, balances from all accounts etc. By the very nature of summation, these computations can often be computed in parallel. These "totals" are then either utilized to adjust the dependent variables for the next iteration in a simulation, or combined with other totals to form a final report, using a certain control mechanism. When large amount of data and computations are involved, parallelism provides a large performance gain over the conventional sequential approach. The UBX™ is one such mechanism to perform the above computations in parallel across multiple computers, i.e., computational nodes. Essential to the claim in this patent application is the way UBX™ synchronizes among all nodes, combines the intermediate results, and handles user tasks in a concurrent fashion.

The number of compute nodes is initially set to meet the database size and simultaneous user requirements necessary to achieve the business users' overall performance goals. The number of compute nodes can be increased at any time to accommodate changes in either performance requirements or number of source systems of record to be catalogued.

The architecture can also be scaled by adding additional levels of SysGovernor

This ability to scale the performance of the system in a cluster parallel manner at the node level allows UBX™ to maintain a far more linear increase in overall throughput than would otherwise be possible by more traditional means. UBX™ has the ability to sort, index, search, manage, stream, buffer and deliver heterogeneous databases in the 1,000 petabyte (10^{18} bytes) range to customers when and where they want them. This is the key challenge for storewidth (storage and bandwidth) and UBX™ delivers it with currently available off-the-shelf Intel based CPU chipware.

The UBX™ is based on original intellectual property including patent 5278987 Virtual Pocket Sorting

UBX™ is an open standard, open source, scalable, distributed, parallel Linux platform. The "open standard protocol system" design means that industry standard hardware can be enhanced with pluggable, embedded proprietary technology for even greater system performance and reconfigurable capabilities. It is a unique combination of software and hardware that easily expands with the growth of the business and its big data. A few of UBX™ characteristic features are as follows:

Unifies Information Across Heterogeneous Databases

UBX™ provides a single view of data from multiple data sources.
These sources include legacy systems, Oracle, and DB2.

Large Datasets Are Processed in Reduced Time

As the size of the big data grows, UBX™ maintains linear scalability. For conventional database systems, the throughput decreases as the size of the database grows. The throughput of the systems is further enhanced since only useable data is stored.

High Speed Computational Engine
UBX™ is able to process queries ten times faster than a conventional database. This is the result of UBX™'s unique indexing algorithm that indexes all fields and requires only one scan of the dataset to process complex computations and sorts. Most conventional databases are hierarchal and require sequential scans of the database slowing down the processing and delivery of the results to the client.

High Data Integrity
UBX™ extracts, transforms, and loads (ETL) data from external databases. UBLoad is able to verify the format of data loaded from external sources. It recognizes format changes and updates instantly. This allows the user to verify and correct the format before the data is processed. Therefore the client consistently receives accurate reporting and analysis.

Web-based GUI Providing Fast Access to Analytic Knowledge
UBX™ interface allows the client to request pre-defined or ad-hoc analysis and reporting. The results can be presented in a standard report format and/or through charts and graphs. The thin layer architecture between the client, UBX interface and the processing engine streamlines processing time and overhead. Results are presented to the client in a minimal amount of time, generally minutes and seconds.

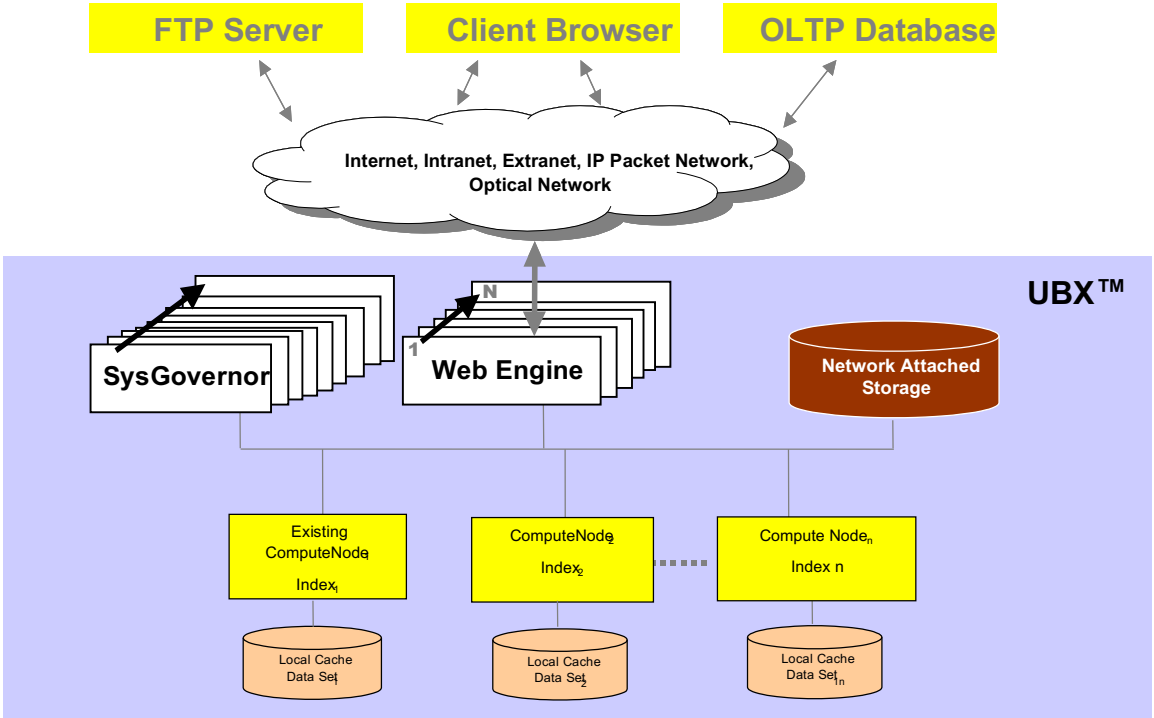


Figure 1 UBX™ System

DETAILED DESCRIPTION AND BEST MODE OF IMPLEMENTATION

The system consists of the UBX™ Hardware platform and the UBX™ System Software

UBX™ Hardware

The hardware components of UBX™ provide a high-speed computing farm in a scalable, parallel cluster architecture that can easily grow to meet the most demanding business requirements. Parallel cluster computing is performed at the node level. Network attached storage expansion is achieved through high speed connections. Data and indexes are mirrored on the local 'cache' for performance and reliability.

We have several system classes to meet our customer's evolving needs. Each system class includes a suite of applications appropriate to the customers operations. The UBX™ system hardware is built from industry standard components to achieve the best price performance ratio. The new system class products are:

Desktop Server Class: This system class is designed for an advanced or "power user" with domain expertise.

Department Server Class: This system class designed for a line of business applications with multiple users.

Enterprise Server Class: This system class is designed for use by the Enterprise server-computing farm:

Custom-Built Class: For those customers who need a special configuration with more storage, more processors or different operating systems, we continue to offer our customized hardware configurations and applications.

Embedded System, A Reconfigurable computing platform

UBX™ - A Parallel Analytical Engine

A mechanism of performing data and mathematical computation in a parallel fashion. It is a parallel adding machine at heart with additional control mechanisms.

There are many examples of data- or computation-intensive applications in scientific, engineering, financial, and other various industrial fields. A good majority of these applications can be decomposed into the simple task of adding up numbers, such as errors of estimated vs. actual at all mesh points in a simulation, balances from all accounts etc. By the very nature of summation, these computations can often be computed in parallel. These "totals" are then either utilized to adjust the dependent variables for the next iteration in a simulation, or combined with other totals to form a final report, using a certain control mechanism. When large amount of data and computations are involved, parallelism provides a large performance gain over the conventional sequential approach.

The UBX™ is one such mechanism to perform the above computations in parallel across multiple computers, i.e., computational nodes. Essential to the claim in this patent application is the way UBX™ synchronizes among all nodes, combines the intermediate results, and handles user tasks in a concurrent fashion.

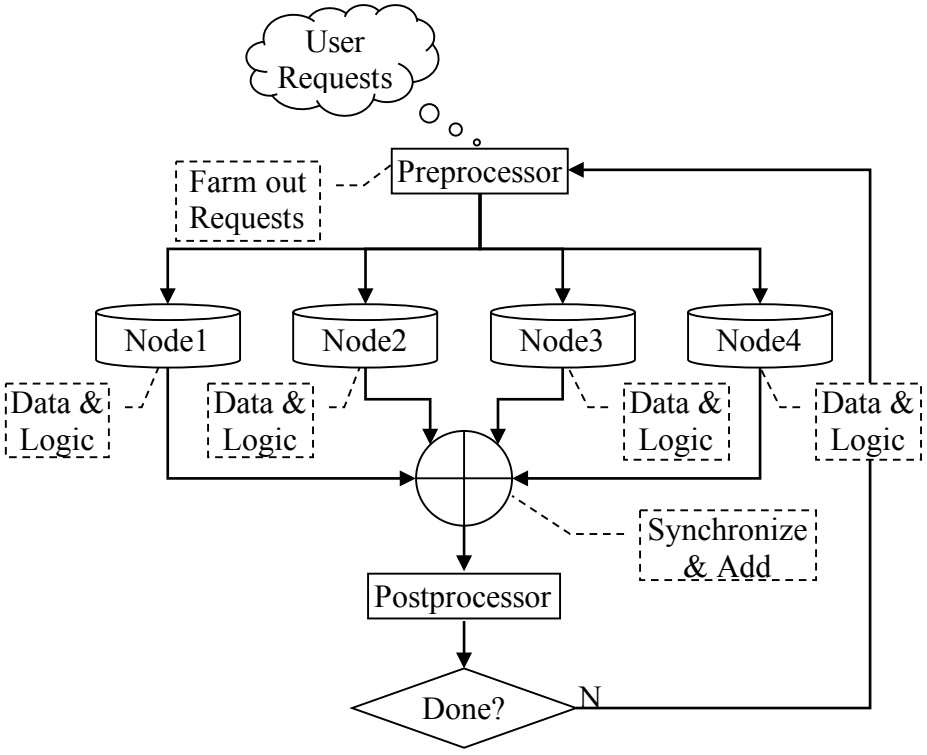


Figure 2. UBX™: user requests execute simultaneously on multiple nodes.

A logical view of UBX™ is shown in Figure 2. In the diagram, four computational nodes run separately on different computers. The data are distributed on each node. The preprocessor formulates and sends computational tasks to all nodes. After each node fulfill its tasks, the intermediate results are synchronized and added before sending to

postprocessor. The postprocessor further manipulates the results and make a decision about the next step.

The following is a physical view of UBX™. Figure 3 shows that the central node, SysGovernor, connects to the computational nodes via network. It provides a user interface for user input and also controls all the computational nodes. It uses a farm of managed threads to manage the communication to and from the nodes. The UBShell that runs on each node handles all communication and data processing task.

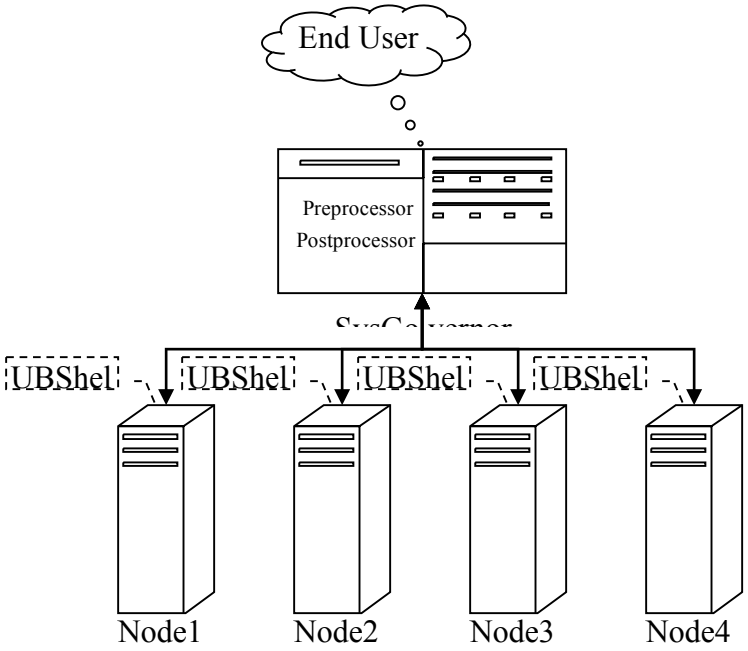


Figure 3. Physical View of UBX™: SysGovernor and nodes are networked together.

From a user's perspective, a typical script is listed in Figure 4. In UBX™, parallelism in data analysis and computation is expressed both implicitly and explicitly.
The `parallel()` explicitly defines portion of the tasks that are executed in parallel on the nodes.
Numbers inside `parallel()`, such as B and map mFreq[], are implicitly added by SysGovernor
Associative memory, such as `mFreq[]`, is explicitly sent to SysGovernor from the node but implicitly combined by the key value, `state`, in our example.

At the end of execution of the sample script
`A = 1`, because it is processed only by SysGovernor
`B = 4`, because it is automatically added by SysGovernor when received from each of the 4 nodes

The table, mytbl, is distributed among all nodes, i.e., each node has its own portion of the data. The associative memory that is created on the table, will thus carry information available for that node only.
The associative memory, `mFreq[]`, is tallied by SysGovernor for each state.
In the `term(g)` block, where "`g`" indicates SysGovernor, the results of tallied `mFreq[]` are printed.

```
 1: A = 1;
 2: parallel(){
 3:   B = 1;
 4:   data(){
 5:         init(){
 6:               x@ = load("mytbl");
 7:               map mFreq(x@, state, curbal);
 8:         }
 9:         term(){ send mFreq[]; }
10:         term(g) {
11:               for_each(state in mFreq[])
12:                     print –obj R@ key, mFreq[state];
13:         }
14:   }
15: }
```

Figure 4. Sample KScript code for UBX™

The implicit parallel mechanism forces user into a mindset of parallel computing framework, whereas the explicit mechanism gives user precise control of how the data are to be computed and combined. The combination of these two controls provides user a unique and powerful way to naturally formulate their problems at hand into parallel computing paradigm on the UBX™ architecture.

In conclusion, UBX™ provides a powerful parallel computing mechanism for data- or computation-centric applications. It does so by implicitly adding values coming from each node while provides explicit control over certain dataflow. For data-centric tasks, the data were first distributed on all nodes. For computation-intensive tasks, logics are divided among the nodes.

A novel way of organizing data and metadata for use in the field of data analytics.

Typical data analysis involves ad hoc query, summation, sorting, and linear table scan. The organization of data and metadata greatly affects the performance and ease of use.

When performing ad hoc queries, many existing analytical tools use a combination of indexing of some frequently used fields and linear table scan on non-indexed fields. One of the reasons that only selected fields are indexed is the large size of index metadata itself. It is also time consuming to create indices.

Another commonly faced problem using conventional tools is the flexibility when creating new fields. When new fields need to be added, it usually requires the whole data (table) be recreated. Therefore, conventional tools provide acceptable and sometimes satisfactory data analysis solutions when the data involved remains relatively small. On the hand, when large amount of data is involved, a better approach is needed.

Our invention uses a novel approach to organize the data and metadata to address the shortcomings of the conventional approach.

We store the data and metadata separately. The data is stored in the most natural form: flat file format. All records and fields have fixed width. All fields are indexed and indices are stored individually. The index data contains the sorting order of corresponding fields.

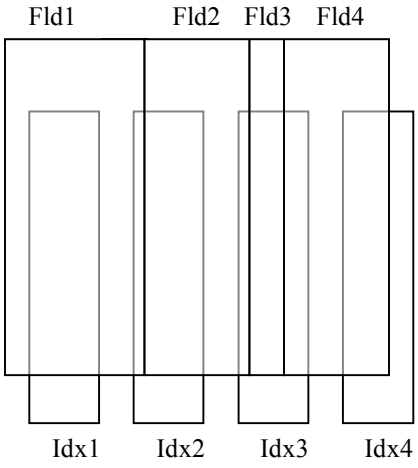


Figure 5. The basic UBFile structure: fixed format data with all fields indexed and stored individually.

The immediate benefits of the above data organization and novel ways to use such data structure are

Ad hoc queries with any combination of fields can be performed without linear table scan

Since we only store the sorting order of the corresponding fields, it does not incur much storage usage. Based on some 1300 different record layouts we have used, covering roughly 3 TB of data and metadata, the metadata occupies about 50% of the total size.
When complex queries are executed, only indices of needed fields are loaded, reducing demand for memory storage
Since we store the data in fixed format, without any additional space reserved in the data file, it is efficient in both storage and retrieval process.
Working together with our indexing mechanism, query results result can be retrieved directly from the data without indirect reference or linear data scan
Once a record is retrieved, individual fields can also be accessed directly without further parsing.
When new fields need to be added, two or more UBFiles can be logically pasted together to form a new UBFile. The component UBFile is self-contained and can be used individually.

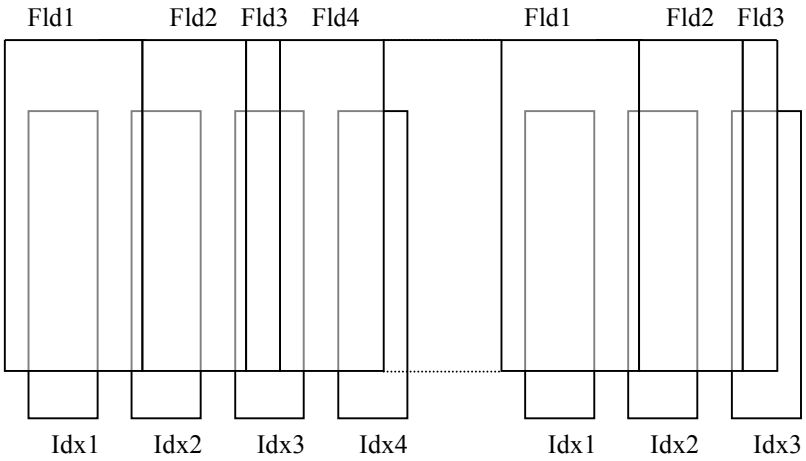


Figure 6. Pasting two UBFiles together logically to form a new UBFile

Multiple UBFiles can also be logically concatenated to for a new UBFile, see Figure 7 in the next page. With the logical concatenation, new data can be added to a logical UBFile without touching the previously existed data.

In short, UBFile is a novel data and metadata organization mechanism that provides space and time efficiency for data management and analysis. It is especially advantageous for operating on large datasets.

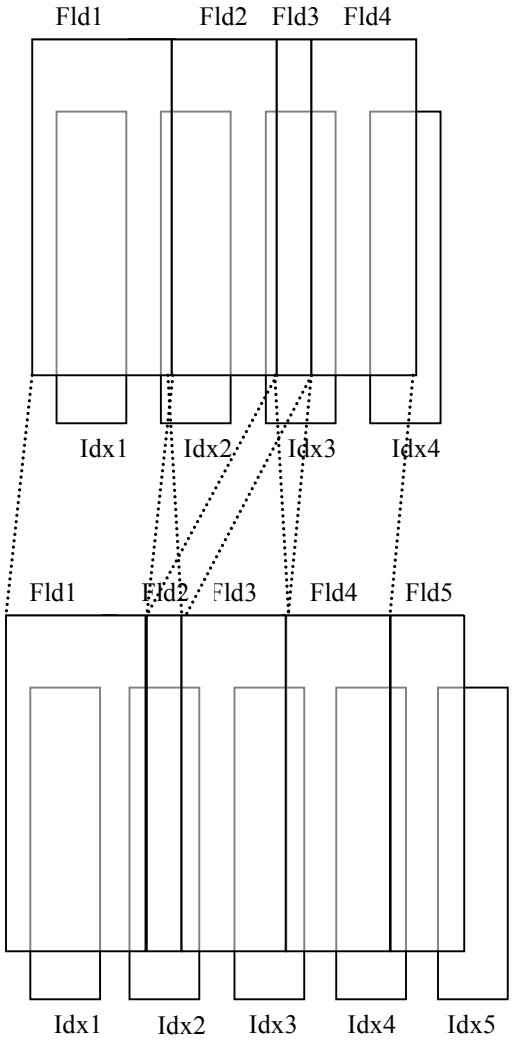


Figure 7. Concatenating two UBFiles together to form a new UBFile.

ASUM - A Multi-Variable Aggregation Technique

A novel way to manage single-pass aggregation on multiple by-variables in any combinations.

Many data analysis and reports are based on aggregation of independent variables within a data file or table. There are many tools that offer ways to perform such aggregations on multiple by-variables. But there has been no solution that provides single-pass and at the same time allowing varying ways of nesting, combining, and concatenating the by-variables. The single-pass and almost limitless ways of combining many by-variables go hand in hand to provide high performance and high capacity report generation over huge amount of data.

With conventional approach, aggregation over each group of by-variables requires a separate pass of data scan, which is not only time consuming, but also makes report generation with many different reports over the same data a very tedious programming task.

Our invention gives user a way to express very complicated aggregation schemes for data analysis and report generation. It allows a user to nest and combine by-variables into sequences and concatenate multiple sequences into a single request. It then translates the request specification into a data structure that describes the relationships of different aggregation sequences. Therefore a single pass of data scan can produce all the reports defined by these sequences. No matter how many aggregations are needed, our approach can always generate them within a single pass of data scan, resulting in substantial timesavings when dealing with large data set.

We use the following data set, Table 1, to illustrate the capabilities of our invention for report generation.

Table 1. Mortgage pool data showing the monthly upb (unpaid principal balance), loan age, and nlns (number of loans within the pool). It also contains pool's static data, such as wac (weighted average of coupon), issue date, pool type, and servicer ID.

poolno	wac	issdt	type	SID	upb	age	nlns
140013	9.346	19830701	30	27	137,256	290	25
141865	8.329	19890401	30	2	132,684	309	13
A00340	11.152	19910501	01	27	28,086	240	8
A00786	10.094	19920901	01	2	79,554	205	57
A00895	10.425	19921201	01	22	6,502	203	28
A01211	8.659	19931101	01	10	17,957	311	22
A01320	9.526	19940401	01	11	2,676	228	7
A01656	10.944	19961001	01	41	12,292	213	3
A01772	9.812	19971201	01	22	14,954	201	16
B00056	10.198	19910701	01	12	117	176	18
B00438	10.111	19940801	03	6	47	178	3
B00557	9.226	19951201	03	7	954	163	11

The reports compute the following values: the total upb, the average loan age (wala) weighted by upb, loan size (lnsz) which is upb divided by nlns, and number of observations (count) with different combinations of by-variables.

The first report, Table 2, aggregates over pool type and SID. For SID, it groups together SID 2, 6, and 10 as one entity, and the rest of possible SID's as another.

Table 2. Report of pool data grouped by pool type and SID.

Type	SID	count	upb	wala	lnsz
ALL	ALL	12	433,708	271	2,053
ALL	2,6,10	4	230,242	262	1684.96
ALL	OTHER	8	202,836	271	2052.5
01	ALL	8	162,137	223	14.2532
01	2,6,10	2	97,511	223	870.769
01	OTHER	6	64,626	223	1019.73
03	ALL	2	1,001	164	9.7696
03	2,6,10	1	47	178	16
03	OTHER	1	954	163	87
30	ALL	2	269,940	299	7103.69
30	2,6,10	1	132,684	309	10,206
30	OTHER	1	137,256	290	5,490

The second report, Table 3, aggregates over variables wac, issdt, and SID. WAC is a numerical field and aggregation is done over uniform increment. IssDT is another numerical field, but the aggregation is non-uniformly spaced. SID is a low cardinality field and the aggregation is done over certain, yet different from Table 1, combinations of possible values.

Table 3. Report of pool data grouped by wac, issdt, and SID

WAC	IssDT	SID	count	upb	wala	lnsz
ALL	ALL	ALL	12	433,708	271	2,053
ALL	ALL	2,10,41	4	242,486	265	2,027
ALL	ALL	6,22,27	5	186,846	271	2,161
ALL	ALL	7,11,12	3	3,747	207	535
ALL	<=1984	ALL	1	137,256	290	5,490
ALL	<=1984	6,22,27	1	137,256	290	5,490
ALL	1985-1991	ALL	3	160,886	262	1,714
ALL	1985-1991	2,10,41	1	132,684	309	10,206
ALL	1985-1991	6,22,27	1	28,086	240	3,511
ALL	1985-1991	7,11,12	1	117	176	6
ALL	1992-1996	ALL	7	119,981	219	818
ALL	1992-1996	2,10,41	3	109,802	222	1,017
ALL	1992-1996	6,22,27	2	6,549	241	562
ALL	1992-1996	7,11,12	2	3,630	207	535
ALL	>=1997	ALL	1	14,954	201	935
ALL	>=1997	6,22,27	1	14,954	201	935
8- 9	ALL	ALL	1	132,684	309	10,206
8- 9	ALL	2,10,41	1	132,684	309	10,206
8- 9	1985-1991	ALL	1	132,684	309	10,206
8- 9	1985-1991	2,10,41	1	132,684	309	10,206
9-10	ALL	ALL	3	156,167	271	2,053
9-10	ALL	2,10,41	1	17,957	311	816
9-10	ALL	6,22,27	1	137,256	290	5,490
9-10	ALL	7,11,12	1	954	163	87
9-10	<=1984	ALL	1	137,256	290	5,490
9-10	<=1984	6,22,27	1	137,256	290	5,490
9-10	1992-1996	ALL	2	18,911	245	612
9-10	1992-1996	2,10,41	1	17,957	311	816
9-10	1992-1996	7,11,12	1	954	163	87
10-11	ALL	ALL	6	103,850	220	871
10-11	ALL	2,10,41	1	79,554	205	1,396
10-11	ALL	6,22,27	3	21,504	241	562
10-11	ALL	7,11,12	2	2,792	208	683
10-11	1985-1991	ALL	1	117	176	6
10-11	1985-1991	7,11,12	1	117	176	6
10-11	1992-1996	ALL	4	88,779	220	871
10-11	1992-1996	2,10,41	1	79,554	205	1,396
10-11	1992-1996	6,22,27	2	6,549	241	562
10-11	1992-1996	7,11,12	1	2,676	228	382
10-11	>=1997	ALL	1	14,954	201	935
10-11	>=1997	6,22,27	1	14,954	201	935

The two reports, Tables 1 and 2, demonstrates the nested aggregation of different by-variables. It contains by-variables that are uniformly spaced (by wac) and non-uniformly spaced (by issdt) variables. It handles user-defined groupings (by SID) as well. With our invention, the above two reports can be generated with a single scan of the data, especially import when dealing with large amount of data.

UBFile Extension: Table Join Indexing

This extension of UBFile addresses the much needed table join capability at table level. Compared to the current join mechanism via TimeSeries configuration, the join relations are created when the tables are updated through join indexing.

The join indexing mechanism is managed through a set of metadata files (tables).

1. The Join Index: Record Index of the Joined Table

The **Join Index** contains the record index (number) of the joined table as shown in Figure 8.

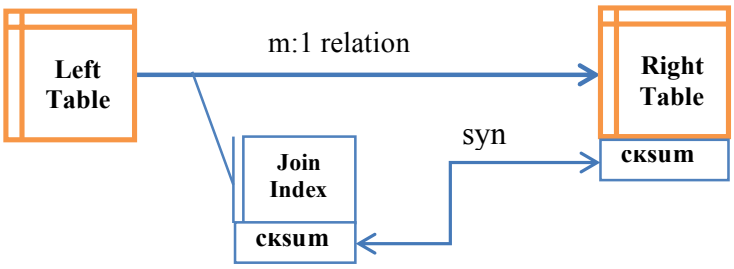


Figure 8. **Join Index.** The left table is joining right table using a Join Index.

In addition, it contains the following:

- joined table name and join key field
- a checksum, **cksum**, of the joined key field in the joined table. The **cksum** is used to ensure the join index is in sync with the joined table.

2. The Metadata: Defining the Join Relationship

The metadata is a file or table defines how a table or a series of tables are joined. Each joining pair consists of the left (local, current, this, etc) and right (remote, historical, other, etc) tables:

1. left table name pattern: the table and key field name or pattern for a series of the left tables
2. right table name pattern: the corresponding right table name and key field
3. lookup order: for fields existing in multiple joined tables, the values are are retrieved from found in the first table following the lookup order. Default lookup order is local table, rules order, and then left, and finally right.

3. The Read Operation

When a field value is read, e.g. via GetFldData(), it looks up the table according the lookup order, and retrieve the field based on joining index.

4. The Indexing Operation

When a left table is updated, it looks up the table-relations metadata to determine how to created the needed join index.

When a right table is updated, it creates a cksum file on the join key field.

5. The Index Syncing Operation

When a right table is updated, all left tables' join indices will be synced, if the cksum on the right join key is changed.

6. Use Cases

The Join Index can be used in the following scenarios

1. Agency PDB Breakout Tables. After pool level data, i.e., gen_pdb and ext tables, are fully populated, the brk_pdb tables only needs to compute and store breakout-specific data, e.g., cbal, nloans, SBals, current LTV (for state breakout), etc. All the other fields are the same as pool level data, perfect candidates for Join Index. We create the Join Index to join the brk_pdb to the gen_pdb. With the Join Index, we can access all the pool level fields, e.g., WAC, WAM, WALA, FICO, SpecPool types, without actually replicating the data in brk_pdb_ext tables.
2. Loan-level database: For loan-level data, each pdb and ext should only store month-specific data, while keeping all the static fields in a single table or virtual vertical tables. The virtual vertical tables consist of monthly originated loans, each table containing a specific origination month. The monthly pdb points to the static pdb via a Join Index.

UBFile Distributed Query and Join: Handling Multiple Relations

Current UBSystem works on data that are distributed across all nodes based on a key field. Various queries and computations are performed over each node's data, and the SysGovernor aggregates intermediate results from the nodes to produce the final results. For applications requiring datasets with multiple keys, data are distributed using different keys. The relationship among these data needs to be handled across nodes' boundaries. To work around this limitation, we merge other relations, e.g., external economic data, static fields, etc., into the single table for analysis. The drawback is the wasted disk space and limited ability for handling diversified data. We address this limitation by the enhancement described here: the distributed query and join.

1. A Whole Virtual Table

The distributed query and join makes use of the centralized storage architecture, where each node has access to all other nodes' data through a high-availability-storage (HAS) complex over high speed network using NFS.

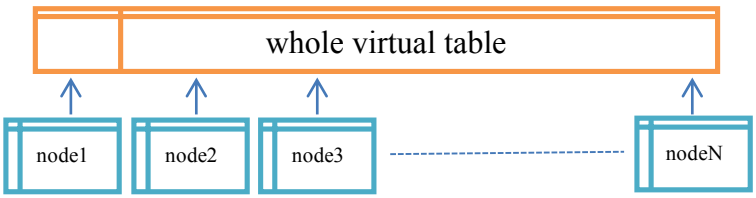
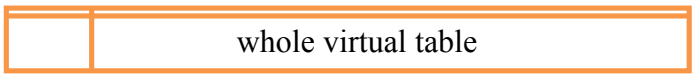


Figure 9. A whole table is a virtual table representing a union of node-component tables visible on the nodes.

As a result of this accessibility, we can define a whole table as a virtual table consisting of all the underlying node-component tables, as shown in Figure 9. A whole table logically concatenates node-component tables as a single virtual table and this table can be accessed from every node.

2. Distributed Query and Join

Once defined, a whole table can be queried and operated just like as a regular table on the node and the query result can be used for subsequent processing independently on each node.



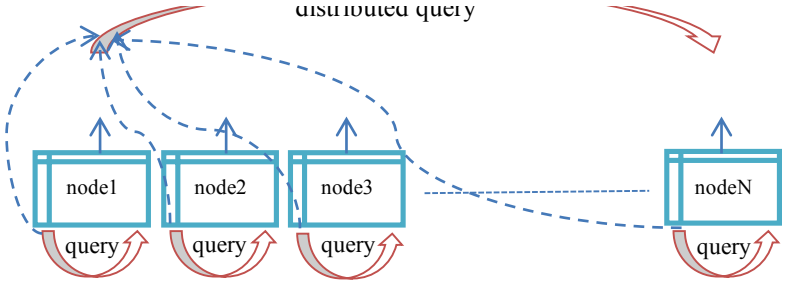


Figure 10. Distributed queries are run on **each node** against the whole table concurrently.

The distributed query result has the same data format as regular node-queries, where reference to each node's location is hidden behind the abstraction of the whole virtual table.
A distributed join is a join that a node-component table joins with a whole table distributed using the UB Table Join Index. It is functionally equivalent to the distributed queries where we nest a node-query with a whole table query. As such we can use either the distributed query or the distributed join based on the specifics of problems at hand. In practice, distributed queries are more efficient and simple to use, which is demonstrated through the use cases to follow, because the query result can be used directly with tables for many types of data analysis.

3. Use Cases

For purpose of illustration, we use MBS CMO data to demonstrate how distributed query and join are used in two test cases involving multiple relationships.
A CMO deal is composed several collateral groups, which is backed by many pools when fully expanded. As shown in Figure 11, a deal-group has a many-to-many, M:N, relationship with the collateral pools.

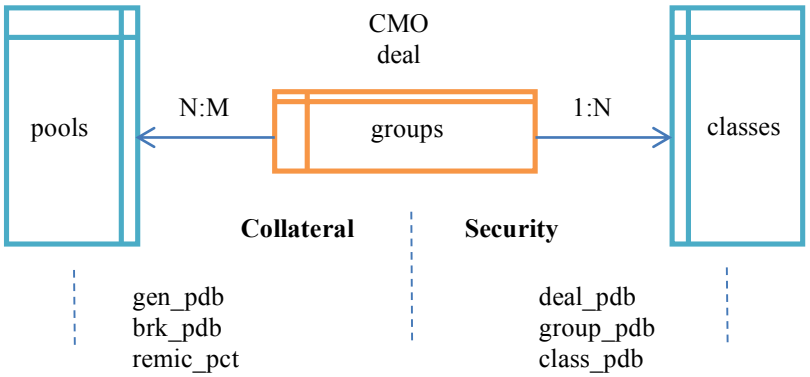


Figure 11. A CMO deal can be modeled as two relations: On the Collateral side, each group maps to M pools and conversely each pool maps to N groups, a M:N relation; and on the Security side, each group maps to N classes.

On the security side, each deal group supplies cash flows to multiple tradable classes, forming a one-to-many, 1:N, relationship.

The collateral pool tables, gen_pdb, brk_pdb, etc, are distributed by pool number field, whereas the security data tables, deal_pdb, group_pdb, class_pdb, etc, are distributed by a different key field, say, the deal-group ID. The collateral composition of a deal-group, i.e., the group-pool M:N relation, is stored in a separate table, remic_pct.

3.1 Reverse CMO Look Up

For a given pool, say 36202FEH5 (G2SF 4636), the goal of this test case is to produce a list of floater and inverse IO classes, e.g., floater class GNR 2010-26 FQ (38376VQR1), and inverse IO class GNR 2010-26 QS (38376VQP5), backed by this pool.

The collateral pool G2SF 4636 is linked to the class GNR 2010-26 FQ and QS via group(s) it backs. Because the group-pool and group-class relations are two separate relations, a distributed query will be required.

The results can be obtained in two steps:

1. Run a node query on the remc_pct table to get the groups backed by the pool
2. Using the groups from step 1's query result, run a distributed query against a whole table of the group-class relation, i.e., the class_pdb.

The final result can be obtained from the class_pdb table using the query output from step 2. Note that the class_pdb itself could be a virtual join table, joining to group_pdb and deal_pdb.

3.2 Selected CMO Bonds Collateral Report

In this case, we are trying to get the security characteristics (e.g., bonds type, current bond balance) of each IO and IIO bonds issued in 2003 through 2012, together with the corresponding collateral info (e.g., prepayment speeds, %Refi, %Servicer, FICO, current LTV).

Similar to the first case, where two queries (node and distributed) are required, except the order of query execution, querying security data first followed by collateral data:

1. Run a distributed query on class_pdb for 2003 – 2012 issuance IO and IIO bonds to get the corresponding deal groups.
2. Using the groups from step 1's query result, run a node-based aggrsum kscript to produce the group-level statistics, e.g., CPR, %Refi, %Servicer, FICO, current LTV, etc.

As a last step, the group-level statistics are merged back into class level entries to produce the final results.

4. Summary and Discussion

With the addition of whole table on the nodes, we can perform distributed queries and joins to link together tables representing different relations, which is functionally equivalent to nested queries and joins in RDBMS.

When complex queries are involved, there may exist different query execution plans leading to the same results. For instance, in test case 3.1, the possible approaches are:

1. a node query on remic_pct table to obtain the groups supported by a given pool, followed by a distributed query on class_pdb to get the floaters and IOs
2. a distributed query on remic_pct table followed by a node query on class_pdb.

Even though they return the same results, but their performance characteristics are different.

This leads a question about how to select the more efficient approach among the many. There is no easy answer to this question, because different queries may have completely different characteristics, demanding a different approach. A simple rule of thumb may just be to choose the first query to produce the smallest query result.

In the case 3.1, perform a node query on remic_pct first in general produces smaller output size. For case 3.2, running distributed query on bond_pdb (0.25M) first may be the right choice, because 1) a whole bond_pdb table is still smaller than a single node remic_pct table (approx. 25M records); and 2) the group-class is a 1:M relation vs the M:N group-pool relation.

5. Variation of the Scheme

Another approach is to reverse the modes of whole vs node table by making the whole table a real one. The node table is just a table containing the disjoint values of the distribution key field, or an alternative table (a non-existing one, mapped through metadata) joining to the whole physical table, using the same Table Join Index.

The advantage is the simplicity by reusing existing components of the current UB.

The drawback could be a slight performance overhead when running analysis on the standalone node tables, i.e., regular queries without using other relations.

Furthermore, with this approach, if we make all the node-table virtual pointing to the whole real table, it will make it easier to adjust nodes. For instance, if we want to change a 36-nodes UB database into 100-nodes, then we only need to re-build the node virtual tables.

UBFile Versioning

1. There are frequent situations where a small portion of a large table needs to be revised
2. a small portion of the table needs to be revised while keeping the original version intact

Current UB handles the above situations by

1. regenerating the whole table
2. creating a separate new table

Which is not efficient in both time and storage consumption.

This enhancement addresses the above issues.

Field-Wise Updates
In this case, only a few fields of a table is updated.

The output() function of data() needs to be updated
1. to allow only selected fields to be updated, either directly replacing the ub1 file
2. to create a new virtual linked table, containing only the updated fields, the ub1 files, while referencing the existing table, which could be a table in a remote UBX instance. A special inter-UBX instance is also implemented to handle node-wise table reference, as described below.

Node-Wise Inter-UBX Operations: Remote Linked Tables
When an instance of UBX has identical number of nodes and containing same data scheme, i.e., same dataset, node-wise inter-UBX operaions can be performed.

On the node, each table in the remote UB instance can directly used without join, as if the remote table is local to this UB instance.

A special table can be a remote link table, similar Linux's symbolic link, which points to a remote node-wise inter-UBX table, except it may contain its own field-wise or row-wise (see below) updates.

Row-Wise Updates
A new UBFile extension is created to handle the update and deletion of rows. This is done by creating an index file that masks rows to be updated or deleted.

A separate component file is added to this virtual file containing the updated rows. The table is thus a vertical file consisting of the masked file and the updated rows.

Materialize Linked Tables
When we want a permanent table without the dependency on the referenced table, we copy and resolve the updates, and merge them into a regular table.

Use Cases
1. A few fields need to be updated with an existing table: use the column-wise update method
2. A few rows need to be updated with an existing table: use the row-wise update method
3. A new version of a table needs to be created while keeping the original table intact, use linked table method

UBX™ WebEngine is a browser-based web application that combines the power of server-side Java 2 Enterprise Edition (J2EE) components, such as XML, JavaMail, Servlets, and JavaServer Pages (JSP). It is a tool for user-driven data analysis and reporting. Features include user authentication, data inspection, support for ad-hoc queries using flexible data selection criteria and record filtering, execution of pre-defined reports, and execution of K-script.

UBX™ Shell
The UB Shell is an application that easily interacts with the UBEngine. It is the user programmable engine within UBX™ Platform. The UB Shell performs sorting; indexing based upon the sort results and joins databases, while minimizing data movement. It runs on all supported platforms. It can run in several different modes: interactive, batch, standalone, and networked. It is implemented using the Object-Oriented technology for flexibility, maintainability, and extendibility.

UB Sort,
The UBX™ uses patented linear sorting algorithms allowing every field to be sorted for indexing and accessed with reduced memory pointer manipulation

UB Index,
The UBX™ uses an unique indexing mechanism allowing every field to be indexed and accessed with reduced memory requirements and pointer manipulation

UBX™ K-Script
UB Script, or K-Script, is a full-featured scripting language supported by *UBX™* to manipulate data, perform data modeling, and implement users business rules. Resembling the syntax of C++ and SAS programming language, it best utilizes *UBX™'s* underlying functionalities and can be further extended to fit users' specific need by using *UBX™*'s C++ API.

UBX™ SysGovernor
The UBX™ SysGovernor uses an industry standard (SAP, TCP/IP or CORBA) compliant communication transport framework providing both synchronous and asynchronous messaging to all of the massively parallel computing UB modules. It oversees the execution of user requests, manages request tracking and controls the operation of the computational nodes where requested tasks are performed. User requested reports are coordinated and notification to the UB WebEngine is initiated as reports become available.

UBX™ API
The UBX™ API enables open integration with UBX™ Platform through a set of C++ , Python & R object classes of mathematical physics libraries. Primary library classes include UBX™ Manager, which provides programmatic access to internal UBX™ functions; CSymbols, which internally stores and tracks all objects within UBX™ and K-Object, which allows users to manage the data type of objects stored in the registry. UBX™ API allows users to develop high performance sophisticated applications for accessing and analyzing large dataset. UBX™ Shell is an application developed using the UBX™ API.

UBX™ Expression

UB Expression is a library containing proprietary mathematical and logical function, which handles complex multi-layered calculations, summations, and business rules.
SumF – utilized to calculate the sum of a field(s) over multiple query results.
AggrSum – used to sum and group results on multiple specified ranges. Results are based on a set of user defined queries, which are performed on a single file or joined multiple files.

UBX™ Mathlib
UB Mathlib is a mathematical physics library containing advanced statistical techniques used to build models:
Non-Linear Least Squares Regression
Maximum Likelihood Estimation
Linear & Non-Linear Logistic Regression
Quantum ElectroDynamics (QED) Field Effect
Deconvolution
Lie Group Symmetry
Differential Geometry of Manifolds
Unified field theory of Gravitation, Electromagnetics, Strong & Weak fields

UBX™ ETL
The UBX™ ETL is a data extraction, transformation, and loading (ETL) tool that loads data from the business data sources, e.g., COBOL data sources, TAP delimited data into the system. The UBX™ ETL tasks are distributed across nodes by the number of records and key field values. It utilizes the data definition files that are defined in the distributed table manager created to find, extract and convert a specified set of records from the 'raw data' which may contain many other record types.

UB Inspector
The UBX™ Inspector (UBI) is a tool for managing, validating and maintaining metadata such as COBOL copybooks and Data Definition Language (DDL) commands from relational databases. It also provides the ability to make changes to the copybook. The UBI unifies different metadata representations by converting them to a standard XML representation.

UBX™ Manipulator
The UB Manipulator (UBM) is the transport engine that actually extracts, combines, and reorganizes data to create the data sets that users have requested.
The previous section explained how the UB Inspector verifies and processes the raw source data into "staged" or "processed" source data. Business decision-makers, using the Metadata Repository, make data requests by selecting from the available processed data. The Metadata Repository then records the user's selections to create and store a global *Source-to-Target Map*.
The UBM reads the Source-to-Target Map to learn about the data a user wants and how he/she wants it organized. The UBM then extracts the needed data from the processed source data, combining data from different source data sets and reorganizing the data if needed according to the user's request into the desired end result: a *normalized, target data set*.

Embedded System, A Reconfigurable computing platform

Background

There are several types of computations and related computer functions that take longer then desired when run on conventional Von Newman architecture computers. One way to address this problem is with many computers running in parallel another approach is to build hardware that can handle the computations much faster. This invention is an example of the second approach.

The advent of Field Programmable Gate Array (FPGA) technology makes practical reconfigurable computing hardware. This invention is a reconfigurable computing platform with several innovative and unique design features.

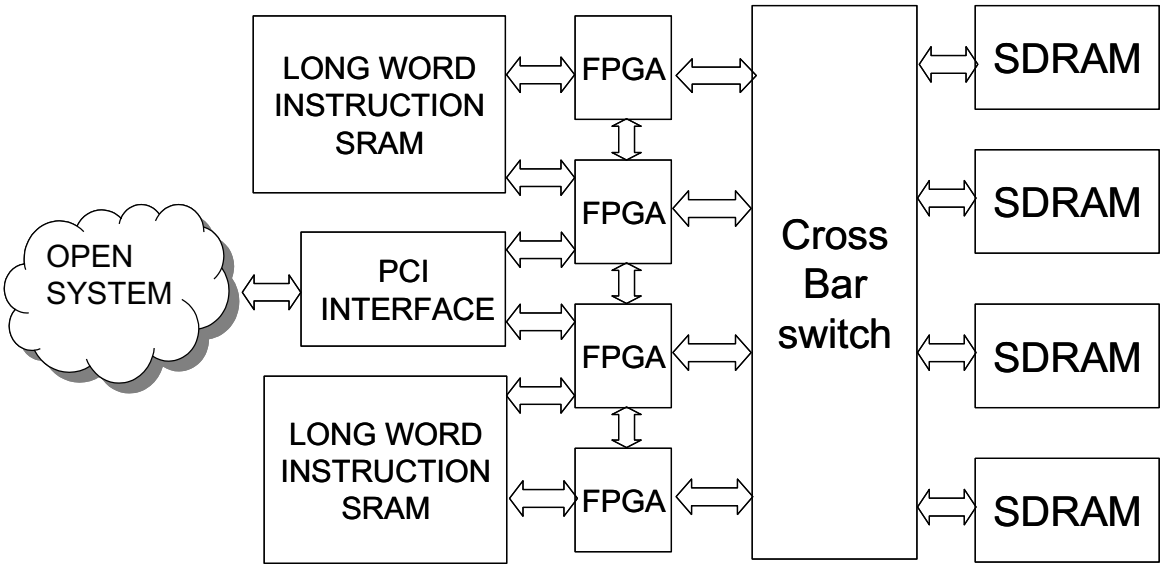
Description

The Embedded System is a hardware system that plugs into a PCI slot on *conventional "open system" host computer* therefore it does not need to duplicate the IO and storage functions already available with off the shelf systems. The open system computer Handles I/O, stores the configuration files for the embedded system, controls the loading the of the appropriate configuration for the tasks at hand, runs the embedded system aware OS or application, and uses the embedded system to process tasks

Unique Features

Multiple FPGA chips.
Multiple Memory arrays.
Crossbar switch connecting any FPGA to any Memory array.
Static Memory arranged is such a fashion to support a very long instruction word Harvard architecture (separate instruction and data memories) configuration.
Wide word reconfiguration path (the entire FPGA system can be reconfigures in a fraction of a second)
Well suited to pipeline based solutions for repetitive computational tasks.

Current Implementation Block diagram and features



64 bit 66 MHz PCI

72 bit data paths (8 bytes + ECC)

4 Xilinx Virtex 1000 Field Programmable Gate Arrays (FPGA)

Up to 12 GB SDRAM, which can be configured as 4 physical memories that can operate independently, in parallel, or as one large memory

A 72 bit by 8 crossbar switch which connects any of the 4 physical memories to any FPGA

Very Long Word Instruction memory -- 216 bit by 256K words of SRAM

Examples of use

Pipeline Configurations
Very Long Instruction Word
UB sort in embedded system

Pipeline Configurations

Ideal for doing the same calculation on many data elements
Performance gains greater than 100 times are practical
Performance is the same for short or long calculations
One data element is completely processed for each pipeline time step.

Pipeline Example: *The Cash Flow Calculation*

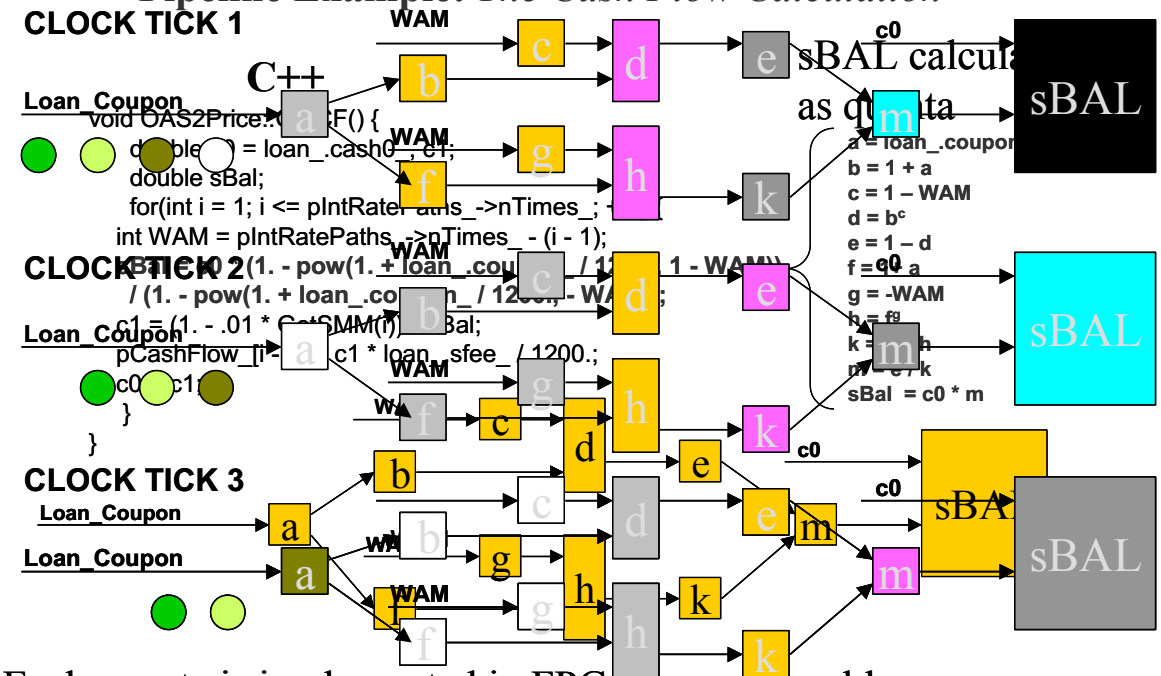
```
void OAS2Price::GetCF() {
    double c0 = loan_.cash0_, c1;
    double sBal;
    for(int i = 1; i <= pIntRatePaths_->nTimes_; ++i) {
      int WAM = pIntRatePaths_->nTimes_ - (i - 1);
      sBal = c0 * (1. - pow(1. + loan_.coupon_ / 1200., 1 - WAM))
       / (1. - pow(1. + loan_.coupon_ / 1200., - WAM));
      c1 = (1. - .01 * GetSMM(i)) * sBal;
      pCashFlow_[i - 1] = c1 * loan_.sfee_ / 1200.;
      c0 = c1;
    }
}
```

1,641 clock ticks for each Iteration of the *for* loop

- The time quanta for the FPGA is equal to 10 clocks of a 1GHZ processor
- For this example the embedded system is about 160 times faster then the C++ open environment
- The rate of completed calculations is independent of the analysis complexity and the data size

Pipeline Example: *The Cash Flow Calculation*

Pipeline Example: *The Cash Flow Calculation*



Each quanta is implemented in FPGA reconfigurable resources

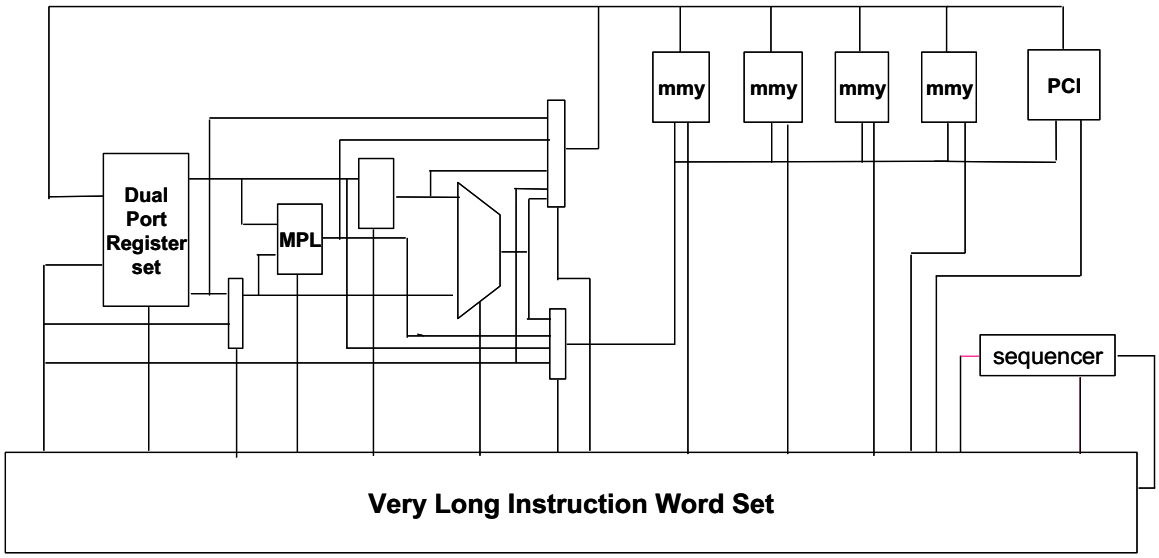
At each time tick the data moves to the next calculation

A data calculation is completed for each time tick

Very Long Instruction Word

Instruction memory is separate from Data memory.
The instruction set memory word is up to 216 bits long, this allows a very rich instruction set.
The system is completely reconfigurable for any instruction set, data path size, function set, constrained only by the amount of FPGA available.
Multiple functions in one instruction cycle.
Full stored program functionality.

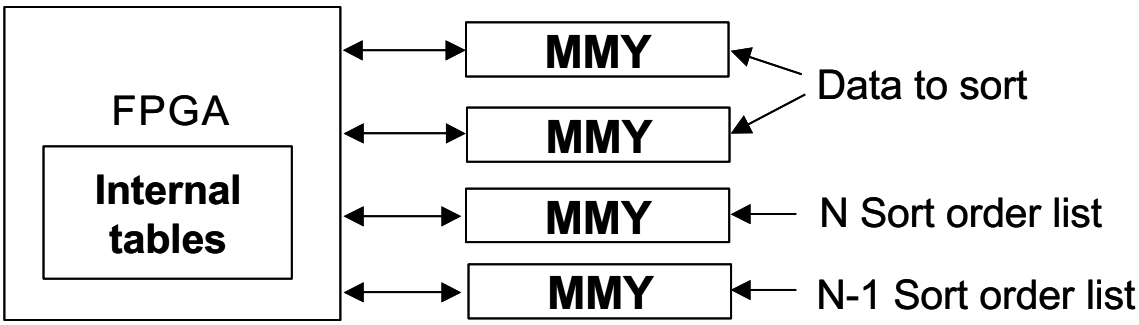
Example: Very Long Instruction Word for custom processor



UB sort in embedded system

Several memory access take place at the same time
All the sort bookkeeping takes place at the same time as the access to the data to be sorted
Linear sort time for number of records and key length
Sort time = Number of records x field length x MMY access time
100,000,000 records x 4 character field = 17 seconds
30 X faster then sort in conventional server (Quad 700 Mhz ZEON)

Quick sort	= 523 seconds
Stable sort	= 601 seconds
Embedded UB sort	= 17 seconds



CLAIMS

Efficient parallel computing
UBX™ uses a system of software to manage parallel computing in an efficient and concurrent fashion by synchronizes among all nodes, combines the intermediate results, and handles user tasks.

Unifies Information Across Heterogeneous Databases
UBX™ provides a single view of data from multiple data sources.
These sources include legacy systems, Oracle, and DB2.

Large Datasets Are Processed in Reduced Time
As the size of the big data grows, UBX™ maintains linear scalability.

High Speed Computational Engine
UBX™ is able to process queries ten times faster than a conventional database. This is the result of UBX™'s unique indexing algorithm that indexes all fields and requires only one scan of the dataset to process complex computations and sorts. Most conventional databases are hierarchal and require sequential scans of the database slowing down the processing and delivery of the results to the client.

High Data Integrity
UBX™ extracts, transforms, and loads (ETL) data from external databases. UBLoad is able to verify the format of data loaded from external sources. It recognizes format changes and updates instantly. This allows the user to verify and correct the format before the data is processed. Therefore the client consistently receives accurate reporting and analysis.

Web-based GUI Providing Fast Access to Analytic Knowledge
UBX™ interface allows the client to request pre-defined or ad-hoc analysis and reporting. The results can be presented in a standard report format and/or through charts and graphs. The thin layer architecture between the client, UBX interface and the processing engine streamlines processing time and overhead. Results are presented to the client in a minimal amount of time, generally minutes and seconds.

Embedded System A Reconfigurable computing platform

Multiple FPGA chips.
Multiple Memory arrays.
Crossbar switch connecting any FPGA to any Memory array.
Static Memory arranged is such a fashion to support a very long instruction word Harvard architecture (separate instruction and data memories) configuration.
Wide word reconfiguration path (the entire FPGA system can be reconfigures in a fraction of a second)
Well suited to pipeline based solutions for repetitive computational tasks.

ABSTRACT

The UBX™ is a matrix computing architecture to perform data- or computation-intensive applications in scientific, engineering, financial, and other various industrial fields in parallel across multiple computers, i.e., computational nodes. Essential to the claim in this patent application is the way UBX™ synchronizes among all nodes, combines the intermediate results, and handles user tasks in a concurrent fashion.