Review of Relational Algebra for Dynamic Distributed Federated Databases

Paul Stone\textsuperscript{1}, Patrick Dantressangle\textsuperscript{2}, Abbe Mowshowitz\textsuperscript{3}, Graham Bent\textsuperscript{4}

\textsuperscript{1}Emerging Technology Services, IBM UK Ltd
Hursley Park, MP137, Winchester, Hants. SO21 2JN, UK
\textsuperscript{2}dantress@uk.ibm.com
\textsuperscript{3}GrahamBent@uk.ibm.com
\textsuperscript{4}abbe@cs.ccny.cuny.edu

Abstract — This paper reviews the coverage of formal Relational Algebra as it applies to distributed, federated databases in varying network topologies. The review shows that a number of Relational Algebra extensions allow distributed relations and federation of heterogeneous database schema. More concrete physical Relational Algebra extensions support access plans for multi-database query processing but lack cost functions dealing with specific network topologies such as scale-free networks, hyper-cubes and Kautz graphs. Statistic gathering techniques are highlighted which allow efficient distribution of database metrics, with the aim of providing optimized query processing.

Keywords — Relational Algebra, Databases, Query Processing, Distributed, Federated, GaianDB, Network Topology

I. INTRODUCTION

Previous papers have described a Dynamic Distributed Federated Database (DDFD) \cite{1} that combines the principles of large distributed databases, database federation, and network topology in a dynamic, ah-hoc environment. A reference implementation of this concept has been produced, named the Gaian Database (GaianDB) \cite{1}.

The GaianDB has been shown to be scalable for simple queries \cite{3} but can be enhanced to optimise complex queries such as joins, aggregate functions and nested queries. Relational Algebra and relevant extensions should be considered to derive the necessary query optimisation algorithms.

We have conducted a review to identify appropriate Relational Algebra extensions including the query optimisation techniques used to determine efficient query access plans.

II. LOGICAL LAYERING OF RELATIONAL ALGEBRA

Relational Algebra may be understood from different perspectives. As a formal system it may be described abstractly in terms of sets and relations; as a software system its description must include implementation (or physical) issues as well.

A. Formal Relational Algebra

The classic Relational Algebra defined by Codd \cite{4} contains purely logical operators which do not model the placement of the data or the details of its storage or retrieval.

B. Physical Relational Algebra

More concrete elaborations of Relational Algebra are used to formulate query access plans. These plans include specific operators depending on the location and storage of the data. Specific operators include IndexScans, which rely on the knowledge that the table is stored with an index, and send/receive operators for transporting intermediate query results between known processing nodes.

III. REVIEW OF FORMAL RELATIONAL ALGEBRA

This section gives a brief background to Relational Algebra and outlines various relevant extensions including aggregate functions, support for null values, multi-sets, multi-databases and meta-data extensions.

C. Mathematical description of a relational database

The Relational Model of data was proposed \cite{3} with the objective of protecting users of database systems from disruptive changes in the representation of data as the database is modified. The Relation Model describes data according to it’s inherent structure without considering the machine representation. As such, the Relation Model provides a foundation on which to build Query languages which are independent of the machine representation and organisation of the data.

Relational algebra, as the name suggests, provides the mathematical tools for manipulating relational databases. The idea of a relational database rests on the observation that a relation is a subset of a Cartesian product. Thus, relational databases may be represented as a subset of the Cartesian product of a collection of sets. More precisely, let $A_1, A_2, \ldots, A_n$ be sets, and for $m \leq n$ suppose $\{i_1, i_2, \ldots, i_m\} \subseteq \{1, 2, \ldots, m\}$ where $i_1 \leq i_2 \leq \ldots \leq i_m$. The Cartesian product contains every possible combination of the individual set members. A "view"
of a database can be represented as a projection of the Cartesian product onto a subset consisting of m of the n coordinates. So, let D be a database represented by a subset of $A_1 \times A_2 \times ... \times A_n$. Then the function $p_i: D \rightarrow A_{i1} \times A_{i2} \times ... \times A_{in}$ defined by $p(a_{i1}, a_{i2}, ..., a_{in})$ is a projection of D onto the $i_{th}$, $i_{th}$, ..., $i_{th}$ coordinates. The n-tuples in D represent the "records" of the database, with a given coordinate corresponding to a "field" in a record; the m-tuples in p(D) represent the elements in a view of the database D. With this representation, operations performed on the database can be modelled quite naturally as algebraic operations on Cartesian products.

Fig. 1 pictures a relation as a subset of the Cartesian product of three sets $A_1$, $A_2$, and $A_3$, representing attributes of cars. $A_1$ consists of four colours; $A_2$ consists of three door styles; and $A_3$ consists of three fuels. The set of two 3-tuples shown in the figure defines a relation as a subset of $A_1 \times A_2 \times A_3$.

A Relational Algebra is defined [4] by providing a number of operations on relations. These consist of standard set manipulation operators (Union, Intersection, and Difference) and additional operators (Cartesian Product, Joins, Projection, Division, and Restriction).

A Relational Calculus is defined [4] as a mechanism to allow the formulation of queries on a database consisting of a number of relations, based on logical predicate calculus.

![Fig. 1 Example of a Relation.](image)

### D. Aggregate Functions

Aggregate functions such as sum, average and other statistical functions are defined [5] as an extension to the relational model. The motivation behind this extension was to provide a theoretical foundation for report generating systems.

Both the algebra and calculus are extended with the concept of aggregate functions and it is shown that the resultant systems are relationally complete.

### E. Nulls

The desire to handle indefinite and incomplete information in databases has led to the addition of "Null" values into the relational model and query languages. Problems arise when considering what constitutes matches for query predicates in relational databases with possible null values.

Yuan and Chiang [6] propose an evaluation algorithm which is sound and complete (i.e. it returns all correct values and only correct values).

### F. Multi-Sets

The canonical relational model does not allow the inclusion of duplicate tuples in a relation. In some database implementations it is useful to allow duplicate records; avoiding duplicate removal can also be a performance advantage in high performance systems.

Grefen and Flokstra [7] present a redefinition of Relational Algebra using multi-sets to allow for duplicates. Multisets comprise of a set and a function mapping the set to a natural number $\geq 1$, such that the function gives the multiplicity of the set element in the multi-set.

Aggregate functions such as count, sum etc. and the Data Manipulation Language are also extended to handle multi-sets.

### G. Multi-Databases

The development of parallel database systems has resulted in the extension of canonical Relational Algebra to handle parallel querying of distributed relations.

Grefen and Flokstra [7] define each relation as potentially split into relation fragments. The paper addresses hash-based horizontal fragmentation, and allocation of tuples to fragments to nodes. New operators are defined to act on the relation fragments.

Copy and split operators perform distribution of data to processing nodes. A Multi-union operator is defined to combine elements into a single multi-set.

An alternative approach to defining parallel relations is taken by Grant et al [8] where atomic relations are composed into multi-relations. For each standard relational operator, an equivalent multi-relational operator is defined to act on multi-relations. These multi-relational operators are MPROJECT, MSELECT, MINTERSECT, MDIFFERENCE, MJOIN and MPRODUCT. A multi-relational calculus is also defined to manipulate multi-relations.

Comparison of these two approaches centres on the approach to data partitioning and referential integrity. The definition of a multi-relation constructed of relations introduces problems with referential integrity constraints. Each individual relation must contain a referentially complete set of data and a relation will not be able to refer to tuples in another relation. In contrast, the approach of defining a relation from relation fragments fits better with a flexible model of referential integrity constraints. Tuples in one
relation fragment can refer to tuples in another relation fragment without contravening referential integrity.

H. Meta Data

The development of federated databases produced the need to manipulate similar data from heterogeneous databases, where the data could have different logical and physical representations.

A Federated Interoperable Relational Algebra (FIRA) is defined by Wyss [9] which allows queries to be formulated and evaluated across a set of partitioned relations. Operators are included which allow the manipulation of relation meta-data, and the mapping of relations between different representations of semantically similar data.

I. Coverage of Dynamic, Distributed, Federated Databases

We have seen that the canonical Relational Algebra has adequate extensions to cover the distribution and federation of data across multiple heterogeneous databases. There are no operators at the formal level that deal with specific network topologies, but this level expressly avoids dealing directly with the physical representation and placement of data, so this the lack of network details is appropriate, and is covered at the Query Processing level.

IV. REVIEW OF QUERY PROCESSING

The optimization of queries has been a major area of activity for the developers and vendors of databases. Framework architectures and optimization techniques have been developed which allow the efficient evaluation of many queries.

We now review the coverage of query processing techniques with respect to Dynamic, Distributed, Federated Databases.

J. Query Processing Overview

D. Kossmann produced a review of Distributed Query Processing in 2000 [10]. This paper provided a comprehensive overview of the query processing techniques being used to implement distributed database systems.

The prevailing architecture for query processing was defined [11] to take a query and transform it in a number of steps into an optimized “access plan” which is then refined and then evaluated by the database management system. This process is illustrated in Fig. 2.

![Fig. 2. Phases of Query Processing](image)

The Parsing phase transforms the Query into an internal representation for convenient manipulation by the later phases.

The Query Rewrite phase performs a number of general optimisations on the query that are applicable regardless of the state of the database.

The Plan Optimisation phase generates and evaluates a number of alternative access plans (which specify how the query will be evaluated). The evaluation of the plans is performed by assessing the cost of executing each step in the plan. The cost of each step depends on knowing details of the volumes, structure and location of the data being queried.

The Plan Refinement phase further modifies the Query Evaluation Plan for efficient interpretation by the Query Evaluation System.

Finally, the Query Evaluation phase takes the optimised, refined Query Evaluation Plan and evaluates it against the database(s).

K. Query Algebra

The representation of the query alters as it proceeds through the phases.

The incoming query is a logical description of the data required and could be expressed in a relational algebra or calculus form.

The query processing phases transform this logical description into equivalent forms more convenient for processing, and then generate, evaluate and select a number of implementation alternatives.

In the query rewrite phase, the query is transformed into a canonical form without the addition of new operators. This canonical form removes unused predicates and simplifies expressions where possible while keeping the semantics of the query unchanged.

Plan Optimisation generates a number of alternatives to evaluate the query; this substitutes physical operators in place of logical operators and also adds additional physical operators to the query evaluation plan.

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Physical Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td>Nested Loop Join,</td>
</tr>
<tr>
<td></td>
<td>Semi-Join,</td>
</tr>
<tr>
<td></td>
<td>Merge-Join,</td>
</tr>
<tr>
<td></td>
<td>Hash Join,</td>
</tr>
<tr>
<td></td>
<td>Bitmap Join</td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>Restrict</td>
<td>Scan,</td>
</tr>
<tr>
<td></td>
<td>Index Scan,</td>
</tr>
</tbody>
</table>

The logical operator “Join” in the query may be evaluated by a number of physical operators: Nested Loop Join, semi-join, merge-join etc. The cost of each of these physical operators is assessed (and at multiple sites of execution) and the most efficient is selected.

The logical operator “Restrict” may be evaluated by an index scan or a full table scan. The Logical Operator is replaced by the most suitable physical operator in the query evaluation plan.

Plans are usually illustrated in a tree structure, showing how a single query is constituted from multiple independent elements. These trees may be expressed as a sequence of nested operators [12] where a more rigorous mathematical analysis is required. These representations are illustrated in Fig 3.
L. Query Cost Functions

Traditional evaluation of Query Evaluation Plan (QEP) cost involves an assessment of the CPU and Disk IO required to perform the physical operations of the plan. Distributed database queries bring the consideration of network IO, with a potential target to optimise the use of limited network bandwidth. Use of sensor networks as databases also brings the objective of evaluating and reducing device power usage [13]. The elapsed time per query is also evaluated and in parallel databases, query time is traded off against resource usage and network usage [12].

In most considerations of query cost optimisation, a particular resource is thought to be the most limited, and heuristics are derived to efficiently use that resource. This is usually CPU or network bandwidth, but TinyDB [22] optimises to minimise device power consumption.

A holistic cost optimisation approach could be derived which allows the system to flexibly determine a system component’s capabilities and the resource bottlenecks albeit network, power or CPU and optimise query processing workloads to reduce bottlenecks.

M. Query Processing for Distributed Databases

The major difference when evaluating queries in a distributed environment is the step of Site Selection. The Query Optimisation process is enhanced to include the possibility of performing sub-queries at capable nodes in the distributed database. Send and Receive physical operators are added to the query plans to allow the evaluation of the data transfer cost of the query.

Kossmann [10] outlines the Site Selection technique, considering processing at the client, server site or a hybrid of the two. Extending to this in a multi-level hierarchy, query plan operators may be evaluated at any depth in the hierarchy.

The placement of data across the distributed nodes may be predetermined, and controlled by a partitioning scheme. In this case, the location and/or cardinality of query data may be accurately assessed and used to optimise operations, particularly joins, where the complexity of the join may be reduced by knowing the location of the data [10].

N. Query Optimisation Phases

As previously highlighted in section IV-J, query optimisation is performed in a number of phases. The sequence of these phases is determined by the dependencies between the various evaluations.

Initially the query rewrite phase transforms the query plan with a number of heuristic rules which are deemed to be valid in the large majority of cases. These transformations do not depend on details of data distribution and cardinality, but solely on the specifics of the query itself.

The Query Optimisation phase considers a number of elaborations of the query plan into more and more physical representations, replacing logical query operators with the most optimal equivalent physical operators, deriving a plan for efficient query evaluation. These elaborations are also performed in dependency order so that later elaborations do not invalidate the earlier ones. An example of this dependency ordering is the site selection of single table access plans, followed by the site selection of joins between two tables, followed by joins between three tables etc [10].

Once estimation (or evaluation) of data cardinality and location has been considered, the query refinement phase considers implementation details of the execution environment to simplify query evaluation.

In consideration of when and where to optimise queries, Kossmann [10] writes that query plans may be derived at query compilation time, or aspects of the system state may be used to modify the plan, or plan alternatives may be selected during query evaluation. Some systems will perform a two step optimisation, pre-compiling the table accesses, join operators and ordering, performing site selection at query execution time.

V. REVIEW OF STATISTIC DISTRIBUTION

The accuracy of database statistics is important to achieve efficient query performance. Inaccurate or out of date statistics lead to query optimisation algorithms selecting poor query plans. On the other hand, the cost of accumulating and storing statistics can be high so it may not be practical to keep accurate up-to-date statistics. Schemes for statistic distribution need to be efficient and need to incur a net gain when used to optimise queries.

Some traditional database implementations require the Database Administrator (DBA) to manually control the collection of statistics; these statistics get out of date as time progresses without regular updating.

Automated Statistics Collection [16] attempts to free the DBA from the burden of statistic collection by providing mechanisms to update statistics automatically. Statistics updates are driven by recognising significant changes to tables.
as well as monitoring the actual results of a query (as opposed to predictions of results). In this case, statistics are updated asynchronously so as not to adversely impact query execution performance.

Self tuning databases [17], [18] gather statistics and estimate the costs of alternative database tuning parameters, including the use of table indices, materialized tables and partitions of table data. It is recognised that the selection of an optimal index set is an np-hard problem [17] and addition of possible materialized tables and partitioning expands the possibility space even further, so these papers outline heuristic based search algorithms to select suitable tuning configurations.

In a distributed environment, passing of database statistics incurs a network overhead and the size of the statistic set becomes more significant. Techniques to compress the statistic set include Bloom Filters [19] which provide estimated membership status and Sketch Estimators [20] which provide approximate answers to aggregate queries. Both of these techniques achieve space saving at the expense of accuracy of the statistics.

VI. REVIEW OF DISTRIBUTED TOPOLOGIES

A number of different distribution topologies are considered as part of query optimisation. The routing of data between multiple processing sites incurs a data shipping cost; some query processing algorithms generate plans with alternative query processing sites and seek to evaluate sub-queries at locations that reduce the shipping cost [10]:

Stand-alone databases execute queries within a single Database Management System (DBMS). On multiprocessor machines, this will be optimised to utilise the parallel compute resource available, query processing may be distributed across multiple threads.

Client-Server databases provide compute resources at the client site which may be exploited to perform query evaluation operations.

Partitioned and Distributed databases allow multiple database instances to coordinate query evaluation and to evaluate query plan operators at various locations.

The topology of database instances in a distributed situation may take a number of forms. Any graph structure could in principle be realized in a distributed database environment. At one extreme every database could be connected to every other database thus forming a complete graph $K_n$ on $n$ nodes. This graph structure is generally too expensive to implement so databases are usually connected to each other via paths of varying length. The underlying structures are thus proper sub-graphs of $K_n$. Common examples include star networks in which every database is connected to a central node, bus arrangements in which databases are connected in a line, and ring structures in which the databases are connected in a cycle.

In a 'freely' evolving network such as the Gaian database, the topology is unpredictable. If preferential attachment is adopted as the growth protocol, for example, the network diameter will be bounded by the log of the number of nodes. Such a protocol may also restrict the degree of each node to a specified maximum. However, the particular paths connecting pairs of nodes are entirely unpredictable with preferential attachment.

Growth protocols may be designed to allow for approximating networks with a specified graph structure. Two graphs in particular have been used extensively to connect machines meant to work together in parallel processing systems. These are the hypercube [14] and Kautz graph [15]. One advantage of engineering an evolving network so as to approximate a known graph lies in having a priori knowledge of the network structure. For example, if the graph is an approximate hypercube, the label of a database node provides information about the nodes to which it might be joined. This
knowledge can be put to good use in solving placement problems in the efficient computation distributed joins. Whenever information has to be obtained from database nodes a, b, c, and d to perform some operation, knowledge of pairwise path lengths between these nodes would have to be determined. This would involve a relatively complex process in the absence of any a priori knowledge, whereas if the graph is known to approximate a hypercube, it is easier to determine the path lengths.

Another aspect to consider is the dynamic nature of ad-hoc network topologies, nodes may actively join or leave the network during the execution of a query. In this case, query optimisation and planning may be best performed in a step-by-step manner, with each node determining locally how to execute the next steps with the knowledge that it has at that moment. Query execution may need re-evaluation when query processing is delegated to a node which then leaves the network.

Consideration of network topology in query processing algorithms varies from a general assumption of the topology type, to a general knowledge high level description of the number of nodes, to a detailed description of the individual node and the connections between them. There is a cost associated with communicating topology metrics and it may be that the cost of communicating this data is higher than the optimisation attained. In this situation, the metrics are not cost-effective and should not be passed. It may be possible to re-engineer a network topology that is causing sub-optimal queries. An analysis of a workload profile could cause nodes to reconnect to form a more efficient network.

Where data transfer has a large overhead, there is potential to move or replicate data to a more convenient location. This may reduce the work required to execute queries, but brings complications of data currency and duplicate detection into consideration.

VII. RELEVANT DATABASE IMPLEMENTATIONS

A number of database implementations have been produced that deal with aspects of distribution, federation and dynamic network topologies. This section highlights a number of relevant examples.

AmbientDB [21] is designed for use in ad-hoc peer to peer networks. The database takes the approach of mapping “abstract global algebra” from the query into “concrete global algebra”. This concrete algebra is then translated into “parallel dataflow execution model” operators, including merge and split functions. Queries are executed as “Dataflows”, parents passing flows of tuples to children and vice-versa. A number of “Wave-plans” are outlined at templates to evaluate various query types including joins and aggregate functions.

TinyDB [22] performs “acquisitional query processing” where queries direct a sensor to gather the relevant data as the query demands. The sensors remain passive until directed, thus reducing the power consumption. This paper describes the maintenance of “Semantic Routing Trees” (SRTs), i.e. data structures to determine the range of attribute data at each node, what data is located where, and so is able to target queries to specific nodes. The benefits of these is compared to the cost of maintaining them, SRTs are created like indices on traditional databases, each node determining whether maintenance of an SRT is beneficial. This will depend on the frequency of data update. The algorithm assumes a static network of nodes, as each node makes assumptions based on the data at connected child nodes.

OGSA-DQP [23] adopts a service orchestration approach to query processing based on grid computing standards. Services are defined which perform query evaluation and these are then orchestrated together by a distributed query service to optimise and execute data queries.

VIII. CONCLUSIONS AND NEXT STEPS

Existing Literature has been identified in the areas of Relational Algebra, query processing, statistics distribution and network topologies.

Further extension of query cost models are required to allow the evaluation of physical Relational Algebra access plans to cope with dynamic scale-free, Kautz and hypercube network topologies in the context of dynamic distributed federated databases like GaianDB.

A model of the costs and benefits involved in statistic distribution is required to allow evaluation of the benefits of these query optimisation techniques.

Further research steps to be undertaken include:

- Consolidation of the relevant Relational Algebra extensions into a canonical definition for this ITA task.
- Modelling of growth and connection management for the differing network topologies. Characteristics of the topologies will be highlighted. This will form a foundation for investigating the impact of different query types in the various topologies.
- Exploration of the association between logical and physical Relational Algebra operators, such as the different join types that can be considered.
- Modelling of the resource costs of different query types and network topology combinations. The landscape of cost will be outlined i.e. which topologies are beneficial for which query types.
- Examination of the overhead of passing statistics describing network characteristics and data cardinality. This will include a cost benefit analysis.

ACKNOWLEDGEMENT

Research was sponsored by US Army Research laboratory and the UK Ministry of Defence and was accomplished under Agreement Number W9111NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry
of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES


