On Creating Efficient Object-relational Views of Scientific Datasets*

Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, Joel Saltz
Department of Biomedical Informatics, The Ohio State University
Columbus, OH, 43210
{krishnan,kurc,umit,jsaltz}@bmi.osu.edu

Abstract

Scientific datasets are often large and distributed in flat files across several storage nodes. Scientists frequently want to analyze subsets of these datasets. A data source abstraction that provides an object-relational view of data while hiding the details of storage and transport mechanisms and dataset layouts is useful in this regard. In this abstraction, Basic Data Sources (BDS) interpret flat files as a set of records and are the building blocks of the view mechanism. Derived Data Sources (DDS) may be built on top of BDSs and provide more complex objects that serve the scientists' needs. The simplest DDS is one that supports a join based view over BDSs. We investigate issues involving building such DDSs for scientific applications and consider distributed versions of the indexed join and the Grace Hash join algorithms. We construct cost models that capture their performance in a restricted space of dataset and system parameters and compare them analytically and experimentally.

1 Introduction

The availability of cheap commodity hardware and faster networks allow for large-scale numerical simulations that can generate terabyte-scale, massive scientific datasets. Such datasets can also arise in imaging or sensor data associated with geophysical sensors, satellites, digital microscopy, and imaging devices used in materials science. Because of their sheer size, scientific datasets are often managed on storage clusters with distributed storage nodes. They are usually stored in flat-files with application specific formats when they are generated from simulations or captured via sensors. A logical object or object-relational view on the data files can benefit scientists in writing analysis algorithms, since the scientists do not need to know about the underlying storage mechanisms and file formats.

One way to offer such an abstraction is to load the data into an object-relational database management system and define views using its mechanisms. This entails a high cost of ingestion for large datasets. As shown in [12], ingesting a 10 GB dataset may take upto 4 hours in systems like Postgres [16]. The alternative is a layered approach [13, 3] wherein each layer provides a more complex view over the objects underneath it. This avoids the cost of ingestion, but requires services to interpret the application-specific file formats and manage views on data files. These services, referred to as extractor functions, can be implemented manually, or generated automatically from layout description languages [17]. An extractor function reads a file segment (also called a chunk) and generates a set of objects or a set of tuples (i.e., an object-relational sub-table). In this paper, we define a *Basic Data Source* (BDS) as consisting of an extractor and a group of file segments. That is, a BDS generates a set of sub-tables. A client may wish to correlate data from different BDSs or aggregate data from BDSs for certain analysis operations. *Derived Data Sources* (DDS), thus, provide more complex views and are layered on BDSs or other DDSs.

Such abstractions, while useful for building applications, can be challenging to optimize. In this work, we present a view creation framework and the services involved in the framework. This paper focuses on a specific kind of DDS that provides a join-based view over BDSs. We compare the Indexed Join(IJ) and Grace Hash(GH) algorithms for implementing a join-based DDS on a PC cluster system. We construct cost models that capture the performance of both algorithms in a restricted space of dataset and system parameters and compare them analytically and experimentally. These cost models will enable us to choose the appropriate algorithm in a given circumstance.

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, CNS #0403342, #CNS-0426241, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

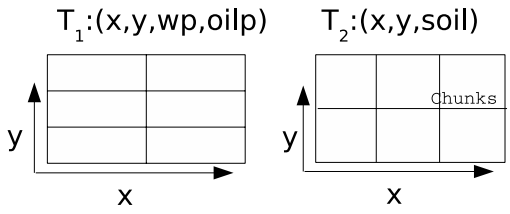

Figure 1. Oil Reservoir Studies Data Example Organization

2 Application Characteristics and Requirements

In this paper, we use oil reservoir management studies [9] as one of the motivating applications for our work. Several other applications such as satellite data processing, image data analysis, and seismic data analysis applications also have similar characteristics.

In oil reservoir management studies, the objective is to understand subsurface properties, long term changes in the reservoir, and identify efficient oil production strategies. This involves generating accurate reservoir models and investigating different placements of injection and production wells. A study would require searching through a space of physical parameters (i.e., reservoir characteristics) and operation parameters (i.e., the number and location of wells) using optimization methods and numerical simulations of the reservoir models. Datasets generated in this fashion are analyzed for economic aspects (e.g., production profits, return on investment) of the reservoir as well as changes in physical properties of the field [9].

Since the domain of parameters is quite large, reservoir simulations can generate terabytes of data. The simulations may be run at different grid resolutions on parallel machines, may partition the grid points in different ways, and simulate different properties of the grid. The output of these simulations may be thought of as a table with coordinate attributes and other properties like saturation of oil, velocity vector, water pressure, etc (a total of 21 attributes for each dataset). The output datasets can be partitioned across multiple storage nodes as a result of parallel execution and written into contiguous file segments in application-specific formats. The file segments are referred to as *chunks* and are the smallest unit of retrieval from the file system. Figure 1 shows a simplified instance of oil reservoir datasets.

Since the data chunks are stored on disk in different layouts, an *extractor* function is needed to interpret the chunks and map them to a standard data structure, referred to here as a *sub-table*. A sub-table may be thought of as a partition of the table structure that comprises the entire dataset. It contains a subset of records and attributes of the dataset table, and methods to iterate through records and attributes in a record.

Metadata information associated with each chunk includes information about which table the chunk belongs to, the location of the chunk in the storage system (i.e., offset in data file) and its size, what attributes it contains, a list of extractors that can read and parse this chunk, and the bounding box of the chunk. The bounding box information contains lower and upper bounds on coordinate and scalar attributes that are stored in the chunk. The lower-left chunk of table T_1 may have the bounding box $[(0, 0, 0.2, 0.3), (64, 64, 0.8, 0.5)]$. The bounding box information is useful in determining which chunks from different tables should be correlated on certain attributes. This bounding box can also be associated with the sub-table produced from the chunk.

To query and analyze a given dataset, a scientist may, for example, wish to access water pressure (wp) and saturation of oil($soil$) attributes of all grid points in reservoir 0. This requires a view definition $V_1 = T_1 \oplus_{xy} T_2$ (i.e., a join between T_1 and T_2 on attributes x and y), on which the above query may be executed. A more complex query could be of the form "Find all reservoirs with average $wp > 0.5$". A view definition, in such cases, may involve aggregation operations such AVG or SUM and ability to define a table as an attribute of another table. Thus, there is a need for a view framework over multidimensional datasets stored in collections of distributed files. In Section 4 we will broach this topic.

3 Related work

There are several works [13, 3, 17] that allow for interpretation of persistent versions of objects. These techniques may be used to generate the extractors we refer to in this paper. Abiteboul [1] applies standard database optimization schemes over structured files. It provides a theoretical foundation for our work. Our work differs from theirs in that we operate in a distributed setting and have to deal with scheduling issues. OGSA-DQP [2] is a framework designed to support object-relational view over disparate data sources. We focus on ways to exploit dataset characteristics to optimize query performance. Relational join algorithms are a well investigated topic in databases and several of them are evaluated in [7, 15]. We have used the parallel version of the Grace Hash join algorithm [8].

I/O optimization in spatial-join operations has been investigated in [18, 4, 5]. Spatial joins are generally implemented as indexed-joins. The Optimal Page Access Sequence (OPAS) involves minimizing the number of page accesses in an indexed-join operation under buffer size constraints. In the general spatial-join operation there is no Grace Hash alternative as the objects being joined are spatial objects. The above works have suggested heuristics to solve the OPAS problem, provided cost models for execution, and compared these models using simulation results.

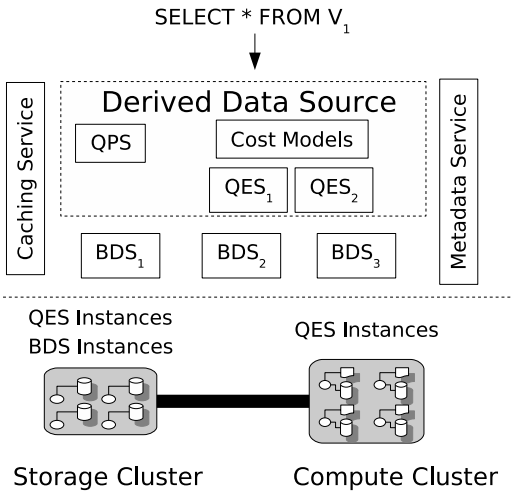

Figure 2. Solution Framework

Our work is complementary in that their algorithms may be used to schedule the sub-table pairs in the IJ algorithms. We compare IJ and GH when the OPAS problem does not affect IJ.

Lee and Chang [10] concentrate on workload balancing and CPU utilization under skew in the index. They claim that hash join always outperforms indexed based join for high number of remote joins for uniform distribution of component sizes (in the page connectivity graph). In our application scenario, all joins are remote joins and component sizes are equal and yet we have identified cases when the opposite is true. We model the impact of dataset characteristics in more detail. Murphy and Rotem [11] have tackled the same problem in the context of multiprocessors with shared memory. They deal with general spatial join problem and do not compare with hash based approach.

4 View Creation

In this work, we target hardware configurations with coupled storage and compute clusters, in which compute clusters are connected by a high bandwidth network or switch to a cluster of storage nodes. Nodes in the storage cluster possess local disks which house data chunks. Nodes in the compute cluster have local memory for caching and local scratch disk space for out-of-core operations, when local memory is not sufficient for operations. With increasing cost-effectiveness of off-the-shelf disks and storage systems built from commodity components, we anticipate that coupled compute-storage cluster platforms will increasingly be deployed at supercomputer centers and research institutions.

An overview of the view creation framework is shown in Figure 2. The role of the Basic Data Source Service is to provide a table view over the application-specific data chunks of a dataset. BDS_i provides a virtual table T_i and is associated with a set of the data chunks. BDS_i, upon receipt of a chunk id j, produces a *basic sub-table* identified by an id (i, j). BDS instances execute on storage nodes and accept requests for sub-tables correspond to local chunks. A BDS accesses information about the location and size of chunks from the MetaData Service.

A Derived Data Source Service provides more complex querying capabilities and is composed of several services. Derived Data Sources (DDS) provide more complex views on the tables exposed by BDSs. The definition of such views may involve selection, projection, aggregation and/or join operations over these virtual tables T_i's. The Query Planning service (QPS) incorporates logic to choose between different Query Execution Systems (QES) based on cost models. A QES implements specific algorithms for operations such as join, projections, aggregations, or selections. The Caching Service can be used by the QES to store and access frequently accessed objects. The MetaData Service stores information about chunks and may also be used by other services to store persistent information.

To illustrate the operation of the framework, we first look at how range queries against BDSs are handled. Suppose a query of the form "SELECT * FROM T_1 WHERE $x \in [0, 256]$, $y \in [0, 512]$" is to be executed. The MetaData Service may be queried using the range part of the query to retrieve ids of all matching sub-tables of T_1. This may be done efficiently using index structures such as R-Trees [6]. Once the sub-table ids are identified, the BDS is asked to generate each of the sub-tables.

Suppose a DDS exposes a view defined by "$V_1 = T_1 \oplus_{xy} T_2$ WHERE $x \in [0, 256]$, $y \in [0, 512]$". In executing a query of a form "SELECT * FROM V_1", it is the task of the QPS to choose the appropriate QES, based on dataset parameters, system parameters and the query, so as to achieve best performance. This requires cost models that can estimate performance of a QES in a specific scenario.

In this work, we concentrate on a DDS that supports equi-joins and range selection operations. We examine two algorithms, the page-level Index Join algorithm (IJ) and the Grace Hash (GH) algorithm and develop cost models for them. We will now briefly describe these two algorithms and how we have implemented them.

4.1 Page-level Indexed Join

Page-level join indices are a special case of spatial join indices [14]. If a relational table is stored as pages in the database management system, a list of page pairs (i, j) such that page i and page j contain at least one record with the same value of join attribute k. When these two tables are required to be joined on the attribute, only these page pairs are checked for matches.

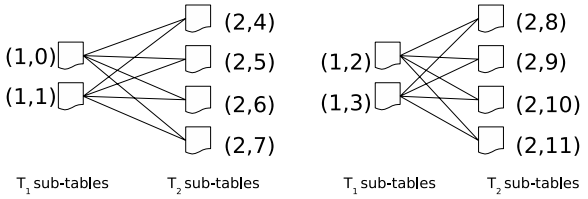

Figure 3. Example Sub-Table connectivity graph (a=2, b=4)

Basic sub-tables can be viewed as pages in a relational database. Each basic sub-table is associated with a lower and upper bound on its attributes. If an attribute is not present in a sub-table, it is assumed to have a bound of $[-\infty, +\infty]$. Sub-tables whose bounds overlap are candidate pairs and they are inserted into a page-level join index. The bounding box of a pair of basic sub-tables is the union of the bounding boxes of each sub-table. Note that this provides an upper bound on the extent of the resulting table from joining the two sub-tables; if the pair of sub-tables is ever joined on some attribute, then this bound can be updated and made tighter. The page-index can be precomputed for common join attributes.

For a dataset, a partitioning of the grid may create different page (sub-table) connectivity graph. Figure 3 shows an example sub-table connectivity graph. A query of the form "`SELECT * FROM` V_1" (where $V_1 = T_1 \oplus T_2$) would be translated to this set of sub-table pairs. Any additional range constraints may be applied at the sub-table level to prune away unwanted edges (and nodes) in the sub-table connectivity graph. Sub-table pairs are joined using an in-memory hash join algorithm and the resultant tuples are generated.

In the IJ approach, each compute node runs a QES instance that receives a pair of sub-table ids to join. The QES instance checks with the local Cache Service Instance to see if either of the sub-tables are present. If not, the QES instance requests for the sub-tables from appropriate BDS instances running on the storage nodes. It then performs a hash join on the received pairs of sub-tables. The QES instance directs the Caching Service Instance to store these recently accessed sub-tables.

4.2 Grace Hash join

An alternative to the indexed-join approach is an output partitioned approach like the Grace Hash algorithm. In this approach, each storage node runs a QES instance that contacts the local BDS instance to retrieve matching sub-tables from the left (inner) table. A hash function (h_1) is used to map records to QES instances, executing on the compute cluster. A compute node QES instance, upon receipt of a record, applies another hash function (h_2) to map the record to a bucket. Buckets are stored on local disks on the compute nodes. The same procedure is repeated with the right (outer) table. Each compute node QES instance then proceeds to join pairs of buckets independently. This is a modification to the algorithm described in [8] to do away with network costs during the bucket joining phase. The number of buckets is chosen so that each bucket fits in memory and an in-memory hash join algorithm is employed. The Grace Hash algorithm is insensitive to the way data is partitioned across the storage nodes, but requires additional I/O operations to split records into buckets.

T	Number of tuples in tables R and S
c_R	Number of tuples in an R sub-table
c_S	Number of tuples in an S sub-table
n_e	Number of edges in connectivity graph
RS_R	Record size of R
RS_S	Record size of S
a, b	Left/right sub-tables in a component
$Net_{bw}(i, j)$	Transfer bandwidth between i storage and j join nodes
$readIO_{bw}$	Disk read I/O bandwidth
$writeIO_{bw}$	Disk read I/O bandwidth
n_s	Number of storage nodes
n_j	Number of joiner nodes
α_{build}	Cost per tuple for building hash-table
α_{lookup}	Cost per tuple for looking up hash-table

Table 1. Dataset and System Parameters

5 Cost Models

In this section, we develop cost models for the IJ and GH algorithms. These models consider both dataset parameters and system parameters which are listed in Table 1. Both algorithms employ an in-memory hash join algorithm as a sub-routine. The in-memory hash join algorithm requires a hash-table be built using the left (inner) relation with the attribute of interest and that the resulting hash table be probed with the records of the right (outer) relation. The cost of computing the hash function and inserting the key into the hash-table is modeled by α_{build}. In our implementation of the hash-table building step, we store a pointer to the relevant record as the value. Therefore, this operation is independent of actual record size. For the same reason, the lookup cost is independent of record sizes and is modeled by constant α_{lookup}.

We will use R and S to denote the left and right tables. We assume that both tables have equal number of rows and the join selectivity is 1 at the record level. This means each record in the left table has only one partner in right table. This assumption is reasonable for the cases in which joins would involve coordinate information such as (x, y). This means for every record in the right table, only one lookup

operation has to be performed. In our cost models, we assume that the cost of extracting a sub-table is much less than the I/O cost of retrieving the corresponding chunk(s) from the I/O sub-system.

5.1 Indexed Join Cost Model

For our datasets, we can assume a sub-table connectivity graph for the view has been precomputed. Since this is a one time process, we do not include the cost of this operation in our cost model. Upon receipt of a query, this index is consulted to identify candidate sub-table pairs which must be scheduled on QES instances in the compute nodes. Independent components of this graph are identified as shown in Figure 3. A component is a connected sub-graph that contains no out-going edges.

The execution cost of IJ is a function of dataset parameters, system parameters and scheduling strategy and cache replacement policy. In developing the cost model, we assume that the tables are regularly partitioned on their coordinate attributes. In our target applications, such a partitioning can be achieved by regularly partitioning the underlying space. This assumption helps us derive average statistics like the average degree of a node in the graph. Our second assumption is that the number of components is much larger than the number of joiner nodes.

We choose the cache replacement policy to be LRU, since this is a reasonable policy in many cases and commonly used. We employ a two stage scheduling strategy. In the first stage, each QES instance in the compute cluster is assigned equal number of components. Then, local id pairs is sorted in lexicographic order of $((i_1, j_1), (i_2, j_2))$ where (i_1, j_1) is the left sub-table id and (i_2, j_2) is the right sub-table id. This ensures that each QES instance in the compute cluster gets the same amount of work.

The third assumption is that if a component consists of a left sub-tables and b right sub-tables, the size of memory in each join node it at least $2 \cdot c_R + b \cdot c_S$. This assumption, with the scheduling strategy, guarantees that no sub-table will be evicted from local cache of a compute node while it is still required for a future computation. Once we have a cost model for this ideal case, it would not be difficult to extend it for cache misses as that will only involve re-retrieving some sub-tables from BDS instances. Here, we compare the best performance of both algorithms.

From the discussion above, we may infer that the execution time of IJ can be broken down into a transfer cost and a CPU cost. The transfer cost entails reading and processing of chunks at the storage nodes and sending requested sub-tables to compute nodes. Since no sub-table is retrieved twice from storage, the transfer cost only depends on the table sizes and network and disk bandwidths. The CPU cost involves building hash-tables on the left sub-tables and probing these with tuples from right sub-tables to produce result tuples. A hash-table is created only once for every left sub-table and cost of creating a hash-table is proportional to the number of records in that sub-table. Probing cost is dependent on the sub-table connectivity graph. If a right sub-table is connected to 2 left sub-tables as in Figure 3, then each record in the right sub-table is involved in 2 lookup operations. In general, the number of lookups is proportional to the degree of nodes representing the right sub-tables and the total number of records of T_1 received by a single joiner node. These are represented by the following equations.

$$
\begin{aligned}
Total_{IJ} &= Transfer_{IJ} + Cpu_{IJ} \\
Transfer_{IJ} &= \frac{T \cdot (RS_R + RS_S)}{min\left(Net_{bw}(n_s, n_j), readIO_{bw} \cdot n_s\right)} \\
Cpu_{IJ} &= BuildHT_{IJ} + Lookup_{IJ} \\
BuildHT_{IJ} &= \alpha_{build}\, T/n_j \\
Lookup_{IJ} &\propto AverageDegreeRSubTable \cdot \\
&\quad NumberofRRecordsinJoinerNode \\
&= \alpha_{lookup}\, \frac{n_e}{T/c_S}\, T/n_j \\
&= \alpha_{lookup}\, n_e c_S/n_j
\end{aligned}
$$

Earlier works have targeted the edge-ratio ($n_e \cdot c_R \cdot c_S / T^2$) as the dataset parameter of importance. Edge-ratio is important to assess the efficacy of scheduling page joins with respect to cache misses. We instead focus on analyzing the computation vs communication cost under certain assumptions.

5.2 Grace Hash Cost Model

Our version of GH is modified so that each joiner node performs its bucket joins independently. This minimizes network costs during the bucket joining phase. The execution time has three components. The transfer component is the same as that in IJ. The amount of data received per node will be the size of several components. All this data cannot be stored in memory simultaneously and hence are written to buckets in local disks. Bucket pairs are then read in and an in-memory hash join is performed on each pair. These costs are proportional to the amount of data processed per joiner node. GH is insensitive to the sub-table connectivity graph. Finally, there is the cpu cost involved in building hash-tables on buckets and probing them. Both these costs are proportional to the total number of tuples per joiner node.

$$
\begin{aligned}
Total_{GH} &= Transfer_{GH} + Write_{GH} + \\
&\quad Read_{GH} + Cpu_{GH} \\
Transfer_{GH} &= \frac{T \cdot (RS_R + RS_S)}{min(Net_{bw}(n_s, n_j), readIO_{bw} \cdot n_s)} \\
Write_{GH} &= \frac{T \cdot (RS_R + RS_S)}{writeIO_{bw} \cdot n_j}
\end{aligned}
$$

$$
\begin{aligned}
Read_{GH} &= \frac{T \cdot (RS_R + RS_S)}{readIO_{bw} \cdot n_j} \\
Cpu_{GH} &= \alpha_{build}\, T/n_j + \alpha_{lookup}\, T/n_j
\end{aligned}
$$

6 Experimental Evaluation

The datasets used in our experiments were generated to exhibit similar characteristics to those of oil reservoir simulation datasets. There are two virtual tables in the dataset. Table T_1 has four attributes $(x, y, z, oilp)$ and table T_2 consists of (x, y, z, wp) where $oilp$ is the oil pressure at a grid point and wp is the water pressure value. The two tables are partitioned along the x, y, and z attribute dimensions. These partitions are distributed along storage nodes in a block-cyclic manner. We varied factors c_R and c_S by varying the partition sizes in powers of 2. Varying partition sizes also affected the sub-table connectivity graph affecting the number of edges n_e. If the size of the entire grid is $[(0,0,0),(g_x,g_y,g_z)]$ and the partition sizes are (p_x,p_y,p_z) and (q_x,q_y,q_z), the size of a component, number of components and number of edges in a component are calculated as:

$$
\begin{aligned}
C &= (max(p_x,q_x), max(p_y,q_y), max(p_z,q_z)) \\
N_C &= (g_x \cdot g_y \cdot g_z)/(C_x \cdot C_y \cdot C_z) \\
E_C &= \lceil \frac{max(p_x,q_x)}{min(p_x,q_x)} \rceil \cdot \lceil \frac{max(p_y,q_y)}{min(p_y,q_y)} \rceil \\
&\quad \cdot \lceil \frac{max(p_z,q_z)}{min(p_z,q_z)} \rceil
\end{aligned}
$$

Queries executed are of the form "`SELECT * FROM` V_1", where $V_1 = T_1 \oplus_{xy} T_2$. Therefore all records in the view have to be generated. The dataset parameters in our cost model are calculated as:

$$
\begin{aligned}
n_e &= N_C \cdot E_C \\
T &= g_x \cdot g_y \cdot g_z \\
c_r &= p_x \cdot p_y \cdot p_z \\
c_S &= q_x \cdot q_y \cdot q_z
\end{aligned}
$$

The hardware configuration used for the experiments is a Linux cluster. Each node has a PIII 933MHz CPU, 512MB main memory, and three 100GB IDE disks. The nodes are inter-connected via Switched Fast Ethernet. In our experiments, we used some of the cluster nodes as storage nodes and some as compute nodes for the join algorithm. We used a maximum of 10 nodes in the experiments.

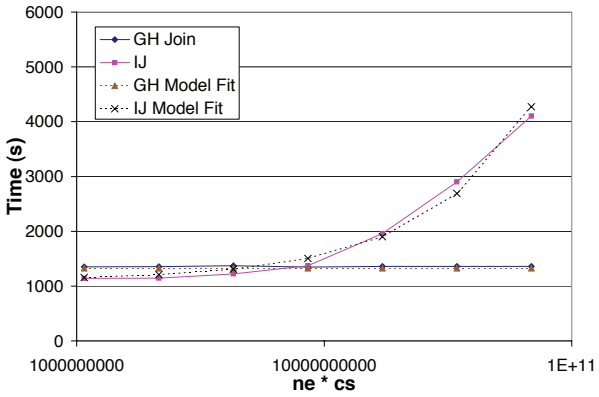

Figure 4. Varying dataset parameter combination $n_e \cdot c_S$

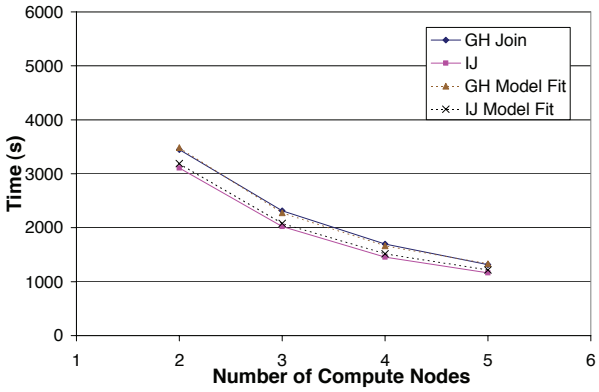

Figure 5. Vary number of Compute Nodes

6.1 Cost Model Validation

The first set of experiments look at the accuracy of our cost models. The dataset characteristics we focused on were $n_e \cdot c_S$, T and record size and the system characteristic we concentrated on was the number of join nodes i.e. n_j.

We first varied $n_e \cdot c_S$ by keeping a constant grid size and varying the partition sizes for T_1 and T_2. Figure 4 shows the execution times of both algorithms under varying $n_e \cdot c_S$. As expected, the $CpuCost_{IJ}$ increases with this factor whereas GH is insensitive to this factor. GH pays the penalty of extra I/O operations because of writing/reading buckets, and this impacts the performance when the value of $n_e \cdot c_S$ is small. This set of experiments was performed with 5 storage nodes and 5 compute nodes. We maintained a constant edge ratio in all of the runs. The edge ratio is calculated as $n_e \cdot c_R \cdot c_S / T^2$.

Figure 5 compares both approaches while varying the number of compute nodes. In this, we chose a dataset with low $n_e \cdot c_S$ value which is why IJ outperforms GH. We observe that the gap in performance decreases as there are

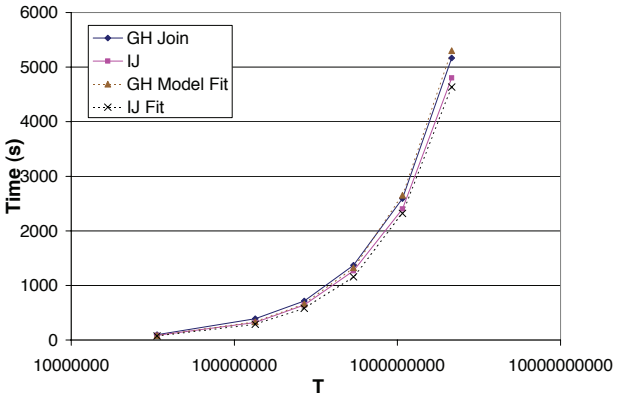

Figure 6. Vary number of tuples

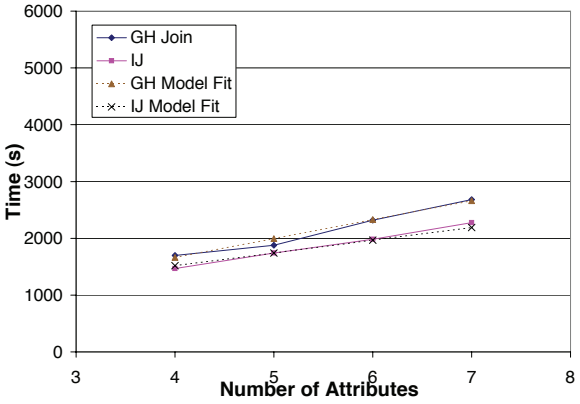

Figure 7. Vary number of attributes

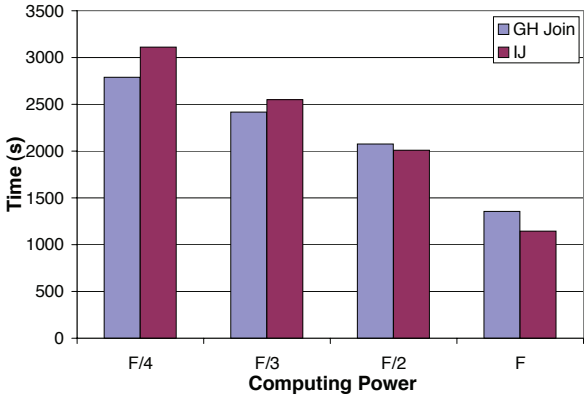

Figure 8. Effect of computing power

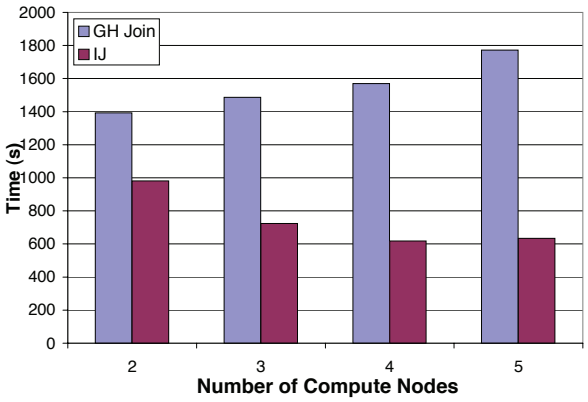

Figure 9. Shared Filesystem

more compute nodes available. This is not surprising as the difference in execution times is inversely proportional to the number of compute nodes.

In Figure 6, we varied the grid size which is equivalent to varying T, the total number of tuples involved in the join operation. We used a maximum of 2 billion tuples in this experiment. As expected, both approaches scale linearly with this factor. Since the difference in execution times also grows linearly, a good choice can make a big difference when tables involved are very large. In Figure 7, we varied the number of attributes in both tables. Each attribute was of size 4 bytes. Varying the record size only affects transfer and read/write costs.

The experimental results show that the models fit actual execution times closely and predict the crossover point (Figure 4) accurately. In the next set of experiments, we investigate the implications of the cost models.

6.2 Discussion of Cost Models

The system parameters α_{build} and α_{lookup} are CPU dependent. We could write these in terms of the unit of processing such as FLOPS, represented by a parameter F: $\alpha_{build} = \gamma_1/F$ and $\alpha_{lookup} = \gamma_2/F$. Here, γ_1 and γ_2 are the number of operations needed to compute each parameter. If we make the additional assumption of $IO_{bw} = readIO_{bw} = writeIO_{bw}$ and the variable $m_S = T/c_S$ representing the number of sub-tables from table S, then we should use IJ when:

$$\begin{aligned} Total_{IJ} &< Total_{GH} \\ Cpu_{IJ} &< Write_{GH} + Read_{GH} + Cpu_{GH} \\ \alpha_{lookup} n_e/m_S &< \frac{2 \cdot (RS_R + RS_S)}{IO_{bw}} + \alpha_{lookup} \\ \frac{IO_{bw}}{F} &< \frac{2(RS_R + RS_S)}{\gamma_2 \, (n_e/m_S - 1)} \end{aligned}$$

Existing trends indicate that processing power increases at a much faster rate than I/O bandwidth. It can also be anticipated that memory space per node on future configurations would be more. Therefore, for the same dataset, IJ will offer more and more improvement over Grace Hash. To demonstrate this, for a particular dataset instance, we varied

the processing rate by repeating the hash building and probing instructions multiple times in both algorithms. To simulate halving the computing power, we performed the building hash-table operation on left sub-tables (or left buckets) twice and carried out two lookups per record. This is admittedly a rudimentary technique, but as Figure 8 shows, the trend predicted by our cost model is accurate. For higher computing powers, we observe that IJ outperforms Grace Hash as expected.

In another experiment (Figure 9), we explored the case when a single Network File System (NFS) storage server serves all the I/O needs of both algorithms. In these experiments, compute nodes are assumed to not have local disks. Results obtained show that GH suffers considerably more than IJ from the shared nature of storage, so much so that increasing the number of compute nodes worsens performance. This is expected as only GH writes buckets to disk. IJ is definitely the better choice under such scenarios.

It should be noted that IJ will not always outperform GH. IJ suffers from the optimal page access sequence (OPAS) problem [18, 4, 5] under high edge ratio values. Intuitively, when the edge ratio is very high, the number of components will be low. This means edges belonging to the same component may be scheduled on different compute nodes causing multiple transfers of the same sub-table over the network. If the number of components were low, then the size of a single component would be large. Therefore, even if a component was scheduled on a single node, there may be local cache misses which might again lead to multiple transfers.

7 Conclusions

We described a framework to create join-based views over scientific datasets. The indexed join (IJ) and Grace Hash join (GHJ) algorithms were analyzed to identify the scenarios in which they may be utilized in view creation. The IJ algorithm is found to be sensitive to the way datasets are partitioned and was able to benefit from it in certain cases. GHJ is, on the other hand, generally impervious to dataset partitions. We developed a suite of cost models, which can be used to choose the appropriate algorithm for a given situation. As part of our future work, we plan to investigate other aspects of view creation, including aggregation operations and caching strategies.

References

[1] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *The VLDB Journal*, 7(2):96–114, 1998.

[2] N. Alpdemir, A. Mukherjee, N. W. Paton, A. A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the grid. In *1st International Conference on Service Oriented Computing (ISOC)*, pages 467–482. Springer Verlag, Nov. 2003.

[3] R. Baxter and et al. Binx - a tool for retrieving, searching and transforming structured binary files. In *Proceedings of UK e-Science All Hands Meeting*, 2003.

[4] C. Y. Chan and B. C. Ooi. Efficient scheduling of page access in index-based join processing. *IEEE Transactions on Knowledge and Data Engineering*, 9(6):1005–1011, Nov./Dec. 1997.

[5] F. Fotouhi and S. Pramanik. Optimal secondary storage access sequence for performing relational join. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):318, Sept. 1989.

[6] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD'84*, pages 47–57. ACM Press, May 1984.

[7] E. P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *VLDB Journal*, 5(1):64–84, 1996.

[8] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[9] T. Kurc and et al. A simulation and data analysis system for large scale,data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, 2005.

[10] C. Lee and Z.-A. Chang. Utilizing page-level join index for optimization in parallel join execution. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):900–914, Dec. 1995.

[11] M. C. Murphy and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–338, 1993.

[12] S. Narayanan, U. Catalyurek, T. Kurc, X. Zhang, and J. Saltz. Applying database support for large scale data driven science in distributed environments. In *Proceedings of the Fourth International Workshop on Grid Computing (Grid 2003)*, pages 141–148, Phoenix, Arizona, Nov 2003.

[13] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[14] D. Rotem. Spatial join indices. In *ICDE*, pages 500–509, Los Alamitos, Ca., USA, Apr. 1991. IEEE Computer Society Press.

[15] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD Conference*, pages 110–121, 1989.

[16] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, Mar. 1990.

[17] L. Weng, G. Agrawal, Ü. V. Çatalyürek, T. M. Kurç, S. Narayanan, and J. H. Saltz. An approach for automatic data virtualization. In *HPDC*, pages 24–33, 2004.

[18] J. Xiao, Y. Zhang, and X. Jia. Clustering non-uniform-sized spatial objects to reduce I/O cost for spatial-join processing. *The Computer Journal*, 44(5):384–397, 2001.