# Skew-Aware Join Optimization for Array Databases

Jennie Duggan[‡], Olga Papaemmanouil[†], Leilani Battle[⋆], Michael Stonebraker[⋆]
[‡] Northwestern University, [†] Brandeis University, [⋆] MIT
[‡] jennie@eecs.northwestern.edu, [†]olga@cs.brandeis.edu, [⋆]{leilani, stonebraker}@csail.mit.edu

## ABSTRACT

Science applications are accumulating an ever-increasing amount of multidimensional data. Although some of it can be processed in a relational database, much of it is better suited to array-based engines. As such, it is important to optimize the query processing of these systems. This paper focuses on efficient query processing of join operations within an array database. These engines invariably "chunk" their data into multidimensional tiles that they use to efficiently process spatial queries. As such, traditional relational algorithms need to be substantially modified to take advantage of array tiles. Moreover, most $n$-dimensional science data is unevenly distributed in array space because its underlying observations rarely follow a uniform pattern. It is crucial that the optimization of array joins be skew-aware. In addition, owing to the scale of science applications, their query processing usually spans multiple nodes. This further complicates the planning of array joins.

In this paper, we introduce a join optimization framework that is skew-aware for distributed joins. This optimization consists of two phases. In the first, a logical planner selects the query's algorithm (e.g., merge join), the granularity of the its tiles, and the reorganization operations needed to align the data. The second phase implements this logical plan by assigning tiles to cluster nodes using an analytical cost model. Our experimental results, on both synthetic and real-world data, demonstrate that this optimization framework speeds up array joins by up to 2.5X in comparison to the baseline.

## 1. INTRODUCTION

Science applications are collecting and processing data at an unprecedented rate. For example, the Sloan Digital Sky Survey records observations of stars and galaxies at nightly rates of 0.5TB to 20TB [22, 33]. Other science projects including the Large Synoptic Survey Telescope and the Large Hadron Collider are even more demanding. As a result, multi-node scalable storage systems are required for these kinds of applications.

Furthermore, skewed data distributions are prevalent in virtually every domain of science [27]. For example, in astronomy stars and other objects are not uniformly distributed in the sky. Hence, telescope measurements have corresponding areas of density and sparsity. Likewise, when marine scientists monitor shipping vessels around US waters [26], there are orders of magnitude more reported vessels near major ports, such as New York, than around less populous coastlines like Alaska.


*SIGMOD'15,* May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723709.

In addition, queries to scientific databases do not resemble the ones found in traditional business data processing applications. Complex analytics, such as predictive modeling and linear regression, are prevalent in such workloads, replacing the more traditional SQL aggregates found in business intelligence applications. Such analytics are invariably linear algebra-based and are more CPU-intensive than RDBMS ones.

The relational data model has shown itself to be ill-suited for many of these science workloads [37], and performance can be greatly improved through use of a multidimensional, array-based data model [34]. As a result, numerous array processing solutions have emerged to support science applications in a distributed environment [5, 6, 9, 10]. Traditionally, these systems do not deal with uneven data distributions. If query operators are not skew-aware, then hotspots and load imbalance between nodes will result, leading to performance degradation. Obviously, this issue will be more serious for more complex operations, such as joins.

Hence, our focus is on optimizing array joins over skewed, multidimensional datasets. Array databases leverage the spatial nature of their data by storing it in "chunks" or multidimensional tiles. Tiles preserve the multidimensional properties of the data and can be leveraged for efficient join processing. As such, the tactics used in RDBMs must be modified for array applications to take advantage of the spatial layout of the array chunks.

Traditionally, join queries on arrays are executed as either a merge join or a cross join [9]. In a merge join the query predicate must include all of the array dimensions of the two operands (which must match), and one array is rechunked to the tiling system of the other. Then, each pair of cells can be merged cell-by-cell. Joins that do not satisfy this requirement (i.e., where the join predicate includes data comparisons not in dimensions) are executed using a cross join. There are several strategies for cross join execution ranging from repartitioning either or both arrays and then executing a merge join to computing the Cartesian product of two arrays. Both of these techniques need to be extended to be skew-aware.

To addresses these challenges we introduce the *shuffle join optimization* framework for the SciDB array data model [10]. It takes advantage of the spatial clustering provided by $n$-dimensional storage tiles to effectively parallelize join execution and minimize data transfers. It also incorporates a range of alternative query plans, using an analytical cost model to identify the best join execution strategy. The cost model includes both network transfer and compute-intensive processing costs in a skew-aware fashion. We focus on equi-joins, but our techniques generalize to other predicates.

Our optimization consists of a logical and a physical planning phase. The *logical planner* picks the join algorithm as well as the subarrays for processing and network transfer. This step is designed to handle any schema alignment requirements among the source and output array and it strives to leverage the preexisting storage organization of multidimensional *chunks*. If this is not possible, the planner makes use of a *dynamic programming optimization* model to compose alternative logical plans. It then identifies the lowest cost method of executing the join and organizing its output array.

The logical planner relies on two primitives, *join units* and *slices*. Join units are non-overlapping collections of array cells grouped by the join predicate (i.e., cells that must be compared for possible matches) and they are built dynamically at runtime specifically for join execution. Working with non-overlapping join units allows us to parallelize the join across the nodes in a cluster, as each join unit can be processed independently. As a result, the join unit defines the granularity at which work is assigned to cluster nodes. To execute the join, we "shuffle" the join units such that matching disjoint sets between the two operand arrays are collocated. These units of network transfer are called *slices* and define the granularity at which data will be moved between cluster nodes.

The *physical planner* assigns join units to cluster nodes. It starts with a *data alignment* step that shuffles the slices associated with each join unit to a single node. It is followed by a *cell comparison* step, that applies the previously selected join algorithm to each join unit. The physical planner uses an analytical cost model to compare competing plans. For each plan, it estimates the query's processing time using the network transfers associated with the data alignment step and the evenness with which the cell comparisons are spread over cluster nodes. We propose a set of algorithms that create these join-unit-to-node assignments, including an integer linear program, a locally optimal search, and a heuristics-based approach.

Our optimization framework is advantageous for two reasons. First, evaluating the query's network transfer and cell comparison costs separately maximizes the flexibility afforded to each physical planner. Second, the physical plans make efficient use of the cluster's sole shared resource, network bandwidth, by reducing data transfer costs. Because CPU is abundant and I/O is readily parallelizable, we optimize the scarcest resource in query processing.

The main contributions of this work are:

- A novel skew-aware join optimization framework for array databases. Our approach identifies the processing unit, data transfer unit, and the join algorithm that minimizes end-to-end query time.
- A dynamic programming model for logical join optimization to identify alternative query execution plans. The optimizer considers a set of array operators that handle schema resolution requirements among data sources and destination arrays. It also incorporates new join algorithms into the array processing model (e.g., hash join), generalizing previous approaches.
- Several skew-aware algorithms to identify the best execution strategy for the logical plan. All of the algorithms use an analytical model that decouples data alignment from cell comparison to discover the join site for each processing unit that will minimize network transfer cost, while balancing the processing across the cluster to prevent hotspots.
- Experiments demonstrating the effectiveness of our techniques. We show a speedup in query latency of up to 2.5X.

We begin our discussion by introducing the array data model and the challenges associated with join processing in Section 2. Section 3 provides an overview of our optimization framework. In Sections 4 and 5, we detail the logical and physical join planning algorithms. Our experimental results are presented in Section 6. Section 7 surveys the related work, and after that we conclude.

# 2. BACKGROUND

In this section, we introduce the basic concepts of the Array Data Model (ADM). We begin with an overview of its logical and physical data representation. Next, we will explore a taxonomy of join processing models. The section concludes with a look at the challenges and opportunities presented by the ADM.

## 2.1 The Array Data Model

In an array database, the storage engine contains a collection of multidimensional objects. Every array adheres to a *logical schema*, which consists of *dimensions*, *attributes*, and a storage layout. Arrays have any number of named *dimensions* that specify its shape. Each dimension is represented by a range of contiguous integer values between 1 and $N$, and the *extent* of the dimension is the number of potential values within this range. Dimensions are ordered, indicating to the optimizer how the query executor will iterate over the array space.

Each array *cell* is identified by its dimension values, or *coordinates*. Cells are analogous to tuples in the relational model. The logical schema contains one or more *attributes*, defining the set of values to store in each cell. Each attribute has a scalar type, such as an integer or float. Each cell may be empty or occupied, and occupied cells contain data for at least one attribute in the array's schema. The database engine only stores occupied cells, making it efficient for sparse arrays.

**Storage Model** Cells are clustered in multidimensional subarrays called *chunks*. Each dimension is divided into logical chunks with a given *chunk interval*, specifying the granularity at which the database accesses cells in each dimension. Therefore, all chunks cover the same number of logical cells, although their stored size is proportional to the count of occupied ones. The database engine uses chunks as its unit of memory, I/O, and network transmission. Each is on the order of tens of megabytes in size, although with storage skew this figure has high variance. Chunk intervals are defined in an array's logical schema. The array's *shape* is defined by its dimension extents and chunking intervals.

Cells in a chunk are *sorted* using C-style ordering on the dimensions, where the cells are sorted one dimension at a time, traversing the innermost one entirely before increasing the outer's value. This enables the DBMS to efficiently traverse and query the array spatially. This arrangement is very limiting for joins, because each cell's chunk position reveals nothing about the attribute values. Therefore, if a join predicate refers to an attribute, the query executor resorts to scanning the entire second array for every cell in the first, performing O($n^2$) comparisons for two arrays containing $n$ cells each. Clearly, there is room for improvement in this approach.

Array chunks are *vertically partitioned* (i.e., each attribute is stored separately). Column stores [1] have demonstrated that for analytical relational databases, vertical partitioning yields an order of magnitude or more speedup over horizontal partitioning. Similar savings in I/O costs are achieved for analytical array databases, because their users often access just a small subset of an array's attributes. This is especially true for joins, where costly data alignment is accelerated by moving only the necessary attributes.

**Example** Figure 1 shows an array with schema `A<v1:int, v2:float>[i=1,6,3, j=1,6,3];`. Array A has two dimensions, $i$ and $j$ (denoted in brackets). This dimension has values ranging from 1...6, inclusive. Both of them have a chunk interval of 3. This schema has two attributes, `v1`, an integer, and `v2`, a float (shown in angle brackets). This sparse array has nonempty cells in just two of its logical chunks, hence it stores just the first and last chunk. The figure displays the on-disk layout of the first chunk, where each attribute is stored separately on disk. When serialized to disk, the first `v1` chunk is stored as `(3,1,1,7,4,0,0)`, in accordance with its C-style ordering.

**Execution Environment** In our work, the array database is distributed using a shared-nothing architecture, where each node hosts one or more instances of the database. Each instance has a local data partition with which it participates in query execution. The

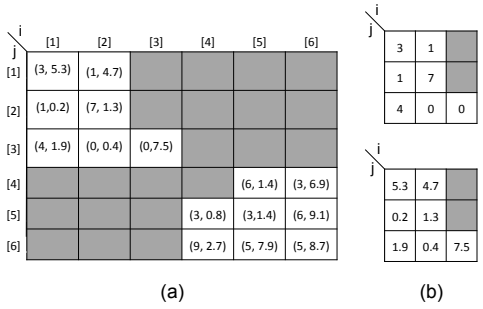


Figure 1: Example: (a) logical array (b) physical layout (first chunk)

entire cluster shares access to a centralized system catalog that maintains information about the nodes, data distribution, and array schemas. A *coordinator* node manages the system catalog, hosting this shared state.

## 2.2 Array Joins

Queries in this data model are written in the Array Query Language (AQL), which is analogous to SQL for relational databases. For example, to filter the array in Figure 1 to `v1` values greater than five in AQL, one would write `SELECT * FROM A WHERE v1 > 5`. Because AQL is declarative, users defer to a query optimizer to plan the execution of any joins. A detailed description of this language is in [9].

Array Functional Language (AFL) is a second way to compose queries in the ADM using nested operators. In SciDB, AQL queries are rewritten internally as AFL queries. The example query above is expressed as `filter(A, v1 > 5)` in this language. Moreover, AFL enables users to seamlessly compose operators, informing the optimizer of their desired operator and execution order. This is crucial for science applications because many of them have workflows where order matters, like transposing a matrix and then multiplying it. We use the declarative AQL to illustrate array queries and AFL to demonstrate their execution plans.

Let us now examine how joins are computed on arrays. Figure 2 shows the join input for all subsequent examples. Our input arrays $\alpha$ and $\beta$ have schemas: $\alpha$<v:int>[i=1,n,k]; and $\beta$<w:int>[j=1,n,k];, respectively.

**Array Joins Notation** To formally reason about array joins, we define an array $\alpha$ as:

$$\alpha = \begin{cases} D_\alpha = \{d_1^\alpha, .., d_n^\alpha\} \\ A_\alpha = \{a_1^\alpha, .., a_n^\alpha\} \end{cases}$$

where $D_\alpha$ is the array's dimension set, and each $d_i^\alpha$ contains the dimension's name, range and chunk interval. $A_\alpha$ specifies the array's attributes and includes their names and types.

To execute a join between arrays $\alpha$ and $\beta$, $\tau = \alpha \bowtie \beta$, one writes the AQL query: `SELECT expression INTO` $\tau$ `FROM` $\alpha$ `JOIN` $\beta$ `ON P` where $P$ is the query's predicate, and it consists of pairs of attributes or dimensions from the source schemas for comparison. In this work we focus on equi-joins, where the predicates take the form of $l_1 = r_1 \wedge l_2 = r_2 \wedge ...$, such that:

$$\begin{array}{l} l_i \in \{A_\alpha, D_\alpha\} \; \forall \, i \\ r_i \in \{A_\beta, D_\beta\} \; \forall \, i \end{array} \tag{1}$$

Hence, the query predicate $P$ is defined as:

$$P = \{p_1, .., p_n\} = \{(l_1, r_1), ..., (l_n, r_n)\} \tag{2}$$

For example, the query `SELECT * INTO` $\tau$ `FROM` $\alpha$ `JOIN` $\beta$ `WHERE` $\alpha$`.i=`$\beta$`.j` has a single predicate: ($\alpha$`.i,`$\beta$`.j`). By formulating the join in this way, the optimizer can infer whether each predicate pair $(l_i, r_i)$ operates on dimensions, attributes, or one of each. It then uses this information to identify the output schema of the join based on the type of the predicate:

- *Dimension:Dimension (D:D)*: This predicate pair matches cells with the same dimension value by merging their attribute values. This operation mostly closely resembles the relational *merge join*. If a cell in either of the input arrays is empty, the corresponding cell in the result is also empty. Figure 2(a) shows the output of the query: `SELECT * INTO` $\tau$ `FROM` $\alpha$ `JOIN` $\beta$ `WHERE` $\alpha$`.i=`$\beta$`.j`. If no specific output schema is defined, the query produces an array matching the shape of its inputs.

- *Attribute:Attribute (A:A)*: This predicate compares the values of individual attributes, and matches for a single value may appear in multiple chunks. Currently, these queries are executed through a *cross join* in the ADM which exhaustively compares all cells in both arrays. We propose a series of optimizations to address these queries in the subsequent sections. Figure 2(b) shows the output of the query: `SELECT i,j INTO` $\tau$`<i:int, j:int>[] FROM` $\alpha$ `JOIN` $\beta$ `WHERE` $\alpha$`.v=`$\beta$`.w`. This produces three output cells. An A:A query produces an output dimensionality that is the Cartesian product of its inputs.

- *Attribute:Dimension (A:D/D:A)*: This matching compares sorted dimension values from one array with the attributes of the other array. As a result, this join type requires an output schema to determine whether to handle the return values as a dimension or attribute. One such query is `SELECT` $\alpha$`.v INTO <v:int>[i=1,n,k, j=1,n,k] FROM` $\alpha$, $\beta$ `WHERE` $\alpha$`.i =` $\beta$`.w`. Its output is shown in Figure 2(c). Current array database implementations do not support this join type. We explain how our optimization strategy extends current techniques to support this join type in Section 4.

By default, the join $\tau = \alpha \bowtie \beta$ is mapped to the output schema:

$$\begin{array}{l} D_\tau = D_\alpha \cup D_\beta - (D_\beta \cap D_p) \\ A_\tau = A_\alpha \cup A_\beta - (A_\beta \cap A_p) \end{array} \tag{3}$$

where $A_p$ and $D_p$ are attributes and dimensions appearing in the predicate. This operation is analogous to a natural join in the relational model. In AQL, all queries can default to the execution plan `cross(`$\alpha$`,` $\beta$`)`, which currently is implemented as an exhaustive comparison of all cells in both inputs and produces the Cartesian product of its inputs. This inefficiency arises because array databases are heavily optimized around the spatial organization of their contents. Hence, many of the operators in a query execution plan rely on their inputs being in a chunk belonging to a schema.

## 2.3 Array Join Optimization Challenges

Recall that current implementations of the array model are very restrictive. A:D/D:A joins that compare dimension values to attribute values are not supported, while joins comparing attribute values in both operands are implemented by cross-joins. This operation sequentially scans the first input array for each cell lookup in the second input array, and therefore does not scale to large input sizes. Apart from eliminating these constraints and providing alternative join implementations, our work focuses on addressing two significant challenges in array join optimization, namely, optimizing the use of expensive schema resolution operations and dealing with skewed data distributions.

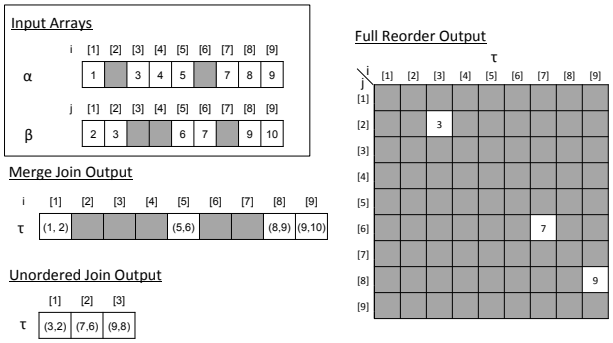


Figure 2: Array joins with varying types of predicates.

### 2.3.1 Schema Resolution

Although the multidimensional storage layout demonstrated in Figure 1 enables array databases to be fast for spatial queries, this specialized layout adds nontrivial complexity to joining their contents. Specifically, the data is ordered by its dimension values and chunks are stored grouped on their position in dimension space. In order to leverage these properties, current array databases implement join queries by executing a merge-join algorithm. Joining arrays with the same logical schema leads to an efficient merge join: since arrays are sorted by the same dimensions, the algorithm combines the arrays by iterating over their chunks in their sort order.

The array merge join has several restrictions on its use. First, it is limited to join predicates that refer to dimensions. Second, the source arrays must have the same dimensions, dimension extents, and chunk intervals. Join queries do not conform to the join's predicates and output shape (i.e., A:D/D:A or A:A) need a reorganization step to use this algorithm. This transformation is known as a *redimension* in the ADM, and it converts any number of attributes to dimensions or vice versa in the source data. For example, if the array in Figure 1 is being joined with array B with the schema `B<v1:int, v2:float, i:int>[j=1,6,3];`, the user would issue an AFL query of `merge(A, redim(B, <v1:int, v2:float> [i=1,6,3, j=1,6,3]))`. The `redim` function converts the attribute `B.i` to a dimension with the same extent and chunk interval as `A.i`. At present, this is usually done manually, but as we will demonstrate it is possible to insert these changes automatically using schema inference.

The redimension gives joins a linear running time, but it redistributes an array and naturally increases the network transfer cost. It also invokes a costly sort operation on the chunks it creates with running time $O(nlog\,n)$, where $n$ is the number of cells in the array. Our framework incorporates a logical planning phase to estimate the benefit and cost of various schema reorganization operators, as well as examine alternative join algorithms that could eliminate the cost of this reorganization.

### 2.3.2 Skew Management

The most efficient way to achieve fast, parallelizable join execution of large arrays is for the matching cells of the two inputs to be hosted on the same server. This allows for parallel cell comparison where each server joins local subarrays. Thus, parallel join execution has two phases: (a) *data alignment* or collocating cells for comparison, and (b) *cell comparison* for evaluating the predicate. Both these steps, however, are sensitive to skew.

**Data Alignment Phase** This phase aims to minimize the transmission of array cells. This phase offers the largest opportunity to improve query performance by bringing sparse or smaller sets of cells to their denser matching counterparts. This is especially profitable when sparsely populated subarrays are joined with dense

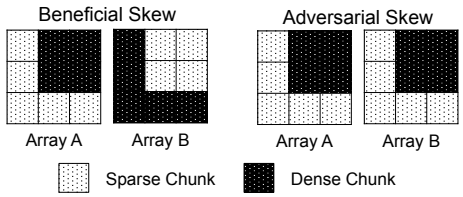


Figure 3: Data distribution types for $A \bowtie B$.

subarrays. We call instances of this imbalance *beneficial skew*, as demonstrated in Figure 3. At the other end of the spectrum is *adversarial skew*, where dense hotspots line up (i.e., dense subarrays need to be matched with dense counterparts), and thus there are limited opportunities to improve the data alignment costs.

Many of our optimizations target beneficial skew, wherein there is a substantial difference in the size of the distributed cells to be joined. This uneven distribution of join cells enables us to minimize the data movement needed to complete the join. One interesting facet of optimizing location-based skew is that simply moving sparse subarrays to the location of their dense counterparts is sometimes not enough. If a query plan transmits all data to a single host, this will create congestion because data is moving over a small fraction of the network links. Therefore sending data to and from all servers will result in better overall performance, even when more array cells are moved. We address this challenge in Section 5.

**Cell Comparison** This step is made fast by maximizing the level of parallelism used when identifying pairs of cells that match a query predicate. Hence, we assign non-overlapping collections of cells to each node for comparison. Here, uneven processing workloads could lead to hotspots and prolong the query execution time.

These two steps of join execution reveal the conflicting goals of shortening the data alignment and cell comparison phases. Striving to distribute the query's work as evenly as possible across nodes may create network congestion, especially if many nodes attempt to transmit data simultaneously. In contrast, if the plan calls for less data transmission, skewed data layouts may result in a disproportionate number of nodes performing the majority of the cell comparisons, prolonging this phase. The physical optimizations discussed in Section 5 address precisely this challenge.

# 3. SHUFFLE JOIN FRAMEWORK

In this section, we begin by walking through the *shuffle join optimization* framework. We first introduce the basic concepts of our framework and the join algorithms implemented therein. We then discuss the query planning process and the plan execution step.

## 3.1 Shuffle Join Terminology

Our shuffle join optimization framework assigns *join units*, or small non-overlapping sets of cells, to nodes in order to make efficient use of network bandwidth and to balance the cell comparison load. Each join unit is in charge of a fraction of the predicate space and cells are assigned to a join unit using either a hash function or range partitioning. Join units enable us to generalize the ADM to a wider range of cell comparison methods.

Cells belonging to a join unit are partitioned over multiple hosts. Hence, each join unit is stored in its source array in one or more *slices*, and an array has at most one slice per host. The data alignment phase transmits all slices that are part of the same join unit to a single node for comparison. When the query is executed, the the cells of its two *sides* are compared; each input array provides one side. In our framework, the *join algorithm* refers to the implementation of the cell comparison phase. Our join algorithms are hash, merge, and nested loop joins.

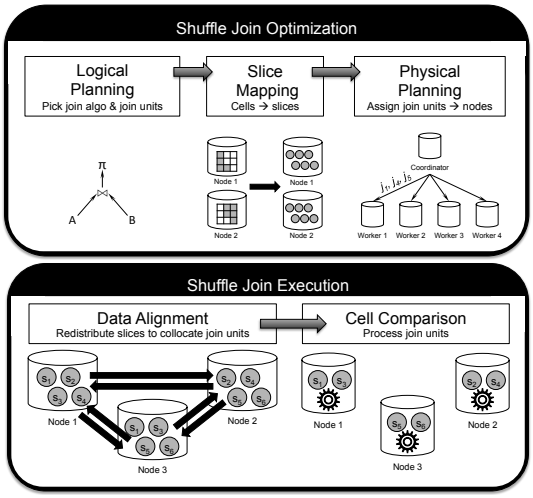


Figure 4: Shuffle join optimization and execution.

## 3.2 Join Algorithms

We now briefly outline the join algorithms available to the logical planner. Each one stems from relational techniques, and they are reviewed here for clarity. All algorithms take as input one join unit, comprised of a set cells for each side of the join.

- **Hash Join**: This algorithm builds a hash map over the smaller side of the join, and it iterates one cell at a time over the larger set of cells, checking each time for a match in the hash map. This join executes in linear time, proportional to its input size, and can operate over unsorted join units.
- **Merge Join**: Merge join requires that the two input arrays have the same shape, that the join's predicates match all of the dimensions but no attributes. It places a cursor at the start of each set of cells, and increments the cursor pointing to a smaller coordinate. If it finds a match between the coordinates at each cursor, it emits the match and advances both cursors. This continues until one or more sets are exhausted. Merge join runs in linear time in relation to its cell count.
- **Nested Loop Join**: This algorithm is similar to the hash join algorithm, where the hash map is replaced with a loop over the smaller set of cells. It has a polynomial runtime but can operate on join units in any sort order.

## 3.3 Shuffle Join Optimization

Figure 4 shows the shuffle join optimization process, which partitions the query planning into logical and physical phases. The *logical planner* prepares the join inputs for fine-grained optimization and collects statistics. The *physical planner* assigns the resulting join units to nodes for fast query execution.

**Logical Planning** This phase analyzes the join predicates, and based on a set of array reorganization operations and a dynamic programming model, identifies candidate execution plans for the query. Using an analytical cost model, it then calculates the cheapest plan. The logical planner also defines join units to complement the join algorithm of each plan. For example, ordered chunks are used as join units to merge joins, and *hash buckets* to hash joins. Note that join units are only created for the query at hand, and are thus a temporary reorganization of the array's contents. Join units are designed to be of moderate size (i.e., tens of megabytes), which supports fine-grained join parallelization without overwhelming the physical planner with options.

**Slice Mapping** Our optimization generates slices, i.e., network transfer units that will collocate matching join units. To achieve this, the optimizer sends a *slice function* to each node, which maps a single cell to its appropriate join unit. Next, each node constructs temporary slices by applying the slice function in parallel to their local cells of the input array. This slice function is also tuned to match the desired join unit. For example, a hash function is used when the join units are hash buckets, and a dimension space when the join units are array chunks.

A slice can also be ordered along a dimension space or unordered. If it is ordered, both arrays will produce slices with the same dimension space and chunk intervals. If it is unordered, a slice may be created with either a hash function or a set of chunking intervals, depending on the source arrays. Both hash and nested loop joins are agnostic to the ordering of their inputs.

**Physical Planning** When a node completes this slice mapping, it reports the size of its slices to the coordinator node of the cluster. The coordinator then passes the slice statistics on to the *physical planner* which uses them to assign join units, $J_i$ in Figure 4, to a single node. It is at the destination node that the slices will be assembled into a single join unit prior to cell comparison. The physical optimizer probes the physical plan search space of join-unit-to-node assignments with an analytical cost model that estimates the overall cost of both the data transfer and cell comparison.

## 3.4 Shuffle Join Execution

Given the final physical plan, the *shuffle join execution* step initiates the shuffle join as shown in Figure 4. It begins with the *data alignment* step. Here, each host iterates over its slices in parallel, sending the ones that are not assigned locally to their destination. In the final phase, the *cell comparison* step, each host combines its slices into join units, and then applies the selected join algorithm to each join unit. During the assembly of the join unit, the execution engine may also preprocess the input, such as sorting the newly formed join unit if it is executing an ordered merge join. Each host processes one join unit at a time, and all work in parallel.

**Shuffle Scheduling** Coordinating this data shuffle can be complicated. If all hosts are exchanging slices at once over a switched network, the data alignment is well-parallelized. On the other hand, if one node is receiving slices from multiple others at once, it may be subject to congestion slowing down the join. Optimally scheduling network transmissions is a combinatorially hard problem [30]. We address this issue greedily by creating a write lock for each host, where the locks are managed by the coordinator node. Hence, when a node seeks to send a slice to node $n_2$, it first attempts to acquire $n_2$'s write lock from the coordinator to gain exclusive access. If the sender cannot acquire the lock, it attempts to send the next slice that is assigned to a node that is not $n_2$. If the sending node runs out of destinations for which it can acquire a lock, it polls for the remaining locks until its transmissions are complete.

# 4. LOGICAL JOIN OPTIMIZATION

In this section, we introduce a logical query optimizer for planning joins in array databases. This planner uses a dynamic programming approach inspired by relational optimizers [32] to produce a set of feasible query plans and to identify the best one to use. Each plan consists of a workflow of AFL operators that a) reorganize the input arrays, if needed, b) specify the join algorithm and the join unit and c) organize the output data to the desired output schema. This planner is designed to extend the ADM to support arbitrary predicates and it improves upon the default plan (a cross join) by identifying more efficient alternatives.

Proposed plans improve upon the cross join in two ways. First, the plan reorganizes lazily, only using expensive schema realignment operators as needed. Second, good plans carefully insert these

| Operator | Cost | Output |
|---|---|---|
| `redim(α, J)` | $n_\alpha + n_\alpha\, log(n_\alpha/c_\alpha)$ | ordered chunks |
| `hash(α, P)` | $n_\alpha$ | unordered buckets |
| `rechunk(α, J)` | $n_\alpha$ | unordered chunks |
| `sort(α)` | $n_\alpha\, log\,(n_\alpha/c_\alpha)$ | ordered chunks/buckets |
| `scan(α)` | – | ordered chunks |

Table 1: Operators for logical join optimization.

reorganizations in the part of the query having the lowest cardinality. This way, if the database needs to make an expensive sort, it does so on a smaller data sets. We analytically model the cost of each plan to compare their estimated duration. In this phase, the cost model works from a single-node model; we relax this assumption in Section 5.

**Join Schema Definition** Joins, like all array database operators, have an output schema, $J$, that denotes the shape of its matches. Hence, for the join $\tau = \alpha \bowtie \beta$, the engine produces $J$ which will be translated to $\tau$ if the two do not match. The join's schema is defined as $J = \{D_j, A_j\}$ and it has a number of properties. Without these features, the join would not be parallelizable.

First, every dimension in the join's schema must appear in an join predicate, $d_i^j \in P \forall i$. The schema dimensions are used to map cells to join units using a conjunction of each cell's dimension values. This mapping is done either by using the chunking intervals like range partitioning to establish the cell's chunk or by applying a hash function to the cell's coordinate. If the join's schema contained a dimension that did not appear in the predicate, this might make cells that are matches appear in different chunks, making the join parallelizable. Note, the dimensions of this schema do not need to cover all of the predicates. If the cells are grouped by a subset of $P$, then all cells that could potentially result in matches will still appear in the same predicate. A related property of $J$ is that it must contain at least one dimension, $d_i^j \neq \emptyset$ or else the join units will no longer be grouped deterministically.

The join schema must produce cells that contain all of the data needed to create the destination schema, $\tau$, and to evaluate the predicate. Hence its attributes are described with $A_j = D_\tau \cup A_\tau \cup P - D_j$. This makes sure that the vertically partitioned database only moves around the attributes that are necessary, but does not perform any unnecessary work.

The dimensions in $D_j$ dictate how cells are meted out into join units if the units are chunks. Here, for each $d_i^j$ the optimizer infers its dimension extent and chunking interval. Because we aim to reorganize the data lazily, if $d_i^j$ is also a dimension in its source or destination schemas, it copies their dimension space opportunistically. If $d_i^j$ is a dimension in $\alpha$, $\beta$, or $\tau$, then the optimizer copies its chunk intervals from the largest one and takes the dimension range from the union of $\alpha$ and $\beta$'s ranges. If one or both sources has $d_i^j$ as an attribute, the optimizer infers the dimension shape by referencing statistics in the database engine about the source data. This usually entails translating a histogram of the source data's value distribution into a set of ranges and chunking intervals.

**Schema Alignment** The join schema may partially overlap with its source and destination shapes. If the source and destination arrays all share a schema, then all the properties above are met, and no schema alignment is necessary.

In the presence of an A:A predicate, the optimizer needs to reorganize both arrays to turn at least the attributes referenced in the predicates into a dimension in the join schema. A:D predicates produce a similar effect, necessitating a reorganization for any array with an attribute referenced during cell comparison. The logical planner confronts a similar issue when determining how to organize the results to the destination schema. If a predicate is an attribute in the final schema, than cells that are part of the same join unit may be stored in different destination chunks.

**Schema Alignment Operators** To address these issues of efficient schema alignment, we have designed a logical planner that uses a set of AFL operators, shown in Table 1. Each of these operators, or a combination thereof, can be inserted before or after a join algorithm to create join units or destination chunks. In this context, the source arrays, $\alpha$ and $\beta$ have $n_\alpha$ and $n_\beta$ cells respectively. Their schemas have chunk counts of $c_\alpha$ and $c_\beta$.

The redimension (`redim`) operator converts one or more attributes of array $\alpha$ into dimensions, producing ordered chunks as its output. The operator iterates over the cells and uses a slice function to assign each cell into a new chunk, an $O(n_\alpha)$ time operation. It then sorts each chunk of $\alpha$ with cost $n_\alpha/c_\alpha\, log(n_\alpha/c_\alpha)$ per chunk. We multiply this by the chunk count, $c_\alpha$ for the total cost estimate.

The `hash` operator creates join units as hash buckets. This slice mapping hashes a source array's cells within $O(n_\alpha)$ time. It produces hash buckets that are unordered and dimension-less. Assignments to this join unit are likely to be sourced from a greater number of chunks, and hence nodes, because the slices are not directly tied to a physical chunk. The presence of more slices enables the physical planner to make its decisions at a finer granularity and hence offers more opportunities for improving the data alignment costs and balancing the cell comparison step.

The `rechunk` operator aligns two source arrays by assigning each cell of one array to the chunk intervals of the join schema. It does not, however, sort the chunks. This may reduce the query execution time if the join is expected to be highly selective, i.e., the query has a small output cardinality. In this case, it makes sense to sort the fewer output cells instead of the input cells. The output join unit will be unordered chunks.

The optimizer may also insert a `sort` operation after the cell comparison. This step can be used to sort the output of a hash join that received its join units from a `rechunk` operator. It has a cost of $n_\alpha\, log(n_\alpha/c_\alpha)$. Lastly, if no schema alignment is needed, then the plan can use a `scan` operator to access the data. This has no additional cost compared to operators that reorganize the data.

**Dynamic Programming Optimization** Once the optimizer has deduced the join schema, it starts to identify plans that will efficiently map the source data to this layout and convert the output cells to the destination schema. We adopt a dynamic programming approach to enumerate the possible plans and their costs. Its steps for $\tau = \alpha \bowtie \beta$ are detailed in Algorithm 1.

```
planList = ∅
for α-Align ∈ (scan(α), redim(α, J), rechunk(α, J), hash(α, P)) do
  for β-Align ∈ (scan(β), redim(β, J), rechunk(β, J), hash(β, P)) do
    for joinAlgo ∈ (hash, merge, nestedLoop) do
      for out-Align ∈ (scan(J), redim(J, τ), sort(J, τ)) do
        p = plan(α-Align, β-Align, joinAlgo, out-Align)
        if validatePlan(p) then
          s = sumCost(p)
          planList.append(s, p)
        end if
      end for
    end for
  end for
end for
return min(planList)
```

**Algorithm 1:** Dynamic programming for logical optimization.

The planner first iterates over the potential steps for aligning $\alpha$ with the schema $J$. It considers all operators (except `sort`) and then progresses on to an inner loop where it does the same with $\beta$. The algorithm identifies one or more join algorithms that can compare cells for the plans. Lastly, it investigates any additional steps needed to adapt $J$'s schema to the output array $\tau$. This step

may call for a redimension to the output schema or a sort if $J$ and $\tau$ share chunking intervals, but the chunk cells are unordered.

Once a potential plan is formed, the optimizer validates the plan. If the plan has a merge join, the validator checks that the inputs are sorted chunks. It also confirms that the output conforms to $\tau$'s schema, precluding a scan after a hash or nested loop join for destination schemas that have dimensions.

If a plan is valid, the optimizer then adds up the cost of its schema alignment step(s) using the equations in Table 1. It also takes into account the cell comparison time, $O(n_\alpha + n_\beta)$ for merge and hash joins and polynomial time, $O(n_a n_b)$ for the nested loop algorithm. It selects the plan with the minimum projected cost.

Although we have simplified our costs in Algorithm 1 to a single node model, only small changes are needed to extend it to a distributed execution environment. If we execute a join over $k$ nodes, cell comparison for hash and merge join takes O$((n_\alpha + n_\beta)/k)$ time. For a redimension of $\alpha$, it takes $n_\alpha/k + (n_\alpha\, log(n_\alpha/c_\alpha)/k$ time. Even when calculating the cost at this finer granularity, the principles of this cost model remain the same. Namely, the planner only reorganizes its inputs as needed and it applies sorting operations to the part of the join with the lowest cardinality, either before or after cell comparison. We address skew in the size of individual join units in the next section.

In this dynamic programming approach, plans that do not call for reorganization, like the D:D example in Figure 2, will be favored over plans that invoke sorts and passes over an entire array to create new join units. This optimizer also speeds up the queries by inserting sorts before or after the join, depending on which has fewer cells. Note that the optimizer does not need very precise estimates of the join's output cardinality to make this assessment. It only needs to know whether or not the output cell count exceeds the size of its inputs to make efficient choices about when to sort the data. Join output cardinality estimation is beyond the scope of this work, but the size of the output, $n_\tau$ may be estimated by generalizing the techniques in [16].

# 5. PHYSICAL JOIN OPTIMIZATION

This section details the second phase of array join planning, wherein the optimizer assigns units of work to individual nodes in the cluster. This step begins with the logical plan created in the last section; it provides a join algorithm and a join unit specification. The latter subdivides the input cells into non-intersecting sets for comparison. With this specification, the query executor create slices for use during the data alignment phase. The physical planner assigns join units to nodes with the aim of minimizing the query's end-to-end latency.

A physical optimizer considers both how evenly the load is balanced across nodes *and* the network costs associated with transmitting data between nodes. Since bandwidth is the only overlapping resource in a shared nothing architecture, the planner carefully rations its use to expedite the data alignment phase. This goal is made complicated if the source data is unevenly distributed over the cluster at the query's start. Here, it may be faster to rebalance the data to maximize the parallelism of cell comparison.

We now introduce an analytical cost model for comparing competing physical join plans. After that, we propose a series of techniques to identify efficient data-to-node assignments.

## 5.1 Analytical Cost Model

To identify fast physical plans, the optimizer needs to qualitatively estimate how assigning a set of join units to processing nodes will impact query completion time. A good cost model approximates the time needed for the proposed data alignment and cell comparison phases of the join by identifying the resource, either network links or compute time, having the most work assigned to it for each of the two phases. This model does not approximate network congestion. Instead, we use the locking strategy presented in Section 3.4, which reduces congestion by only allowing one node to write to a network link at any given time.

To estimate the load on each node, the model calculates the maximum number of cells sent or received by any single node, and multiplies this value by the cost of transmission. The model then calculates the maximum number of cells assigned to each host, and multiplies this quantity by the cost of processing a single cell. By adding the two costs, the first for data alignment and the second for cell comparison, the model estimates the duration of the entire join.

A physical plan consists of $n$ data-to-node assignments, one per join unit, represented with a set of binary variables, $x_{i,j} \in (0, 1)$. If a query has $n$ join units distributed over $k$ nodes, then we say that join unit $i$ is assigned to node $j$ if the binary assignment variable, $x_{i,j}$, is equal to one. Each join unit is assigned to exactly one cluster node, hence all valid plans have the constraint:

$$\sum_{j=1}^{k} x_{i,j} = 1 \; \forall \, i \tag{4}$$

To estimate the time of the data alignment step, the model first considers the number of cells to be sent by each host. For each node, this cost is proportional to the number of cells in its slices that are part of join units assigned to another node. If join unit $i$'s slice on node $j$ has $s_{i,j}$ cells, the cost for sending these cells is:

$$s = \max_{j=[1,k]} \sum_{i=1}^{n} \neg x_{i,j} s_{i,j} \tag{5}$$

Since nodes can both send and receive data across the network at the same time, the model also calculates the host receiving the most data. Let $S_i$ be the total number of cells within join unit $i$, across all nodes. Therefore, a given host will receive no greater than:

$$r = \max_{j=[1,k]} \sum_{i=1}^{n} x_{i,j}(S_i - s_{i,j}) \tag{6}$$

cells. For each slice assigned to a node, its received cells equal to the join unit's total cell count less the ones stored locally. The model quantifies the time of the proposed plan's data alignment as the maximum of the send and receive times, $max(s, r)$.

The model also approximates the plan's cell comparison time. In this step, we estimate the amount of query execution work assigned to each node. This cost is variable, depending on the join algorithm used. For a merge join, it has a cost of $m$ per cell. Here, the join unit's cost, $C_i = m \times S_i$. When a hash join is selected, for each join unit $i$, if we say that its smaller side has $t_i$ cells and its larger one has $u_i$, then the join unit's cost is $C_i = b \times t_i + p \times u_i$. Here, $b$ denotes the build cost of each tuple and $p$ represents the time needed to probe a hash map. This more complicated cost model is borne from the observation that the time needed to build a hash map is much greater than that of probing one.

The model sums up the number of cells assigned to each node and identifies the node with the highest concentration of cells:

$$e = \max_{j=[1,k]} \sum_{i=1}^{n} x_{i,j} C_i \tag{7}$$

If $t$ is the cost of transmitting a cell, the plan's total cost comes to:

$$c = max(r, s) \times t + e \tag{8}$$

where the aggregate for each phase is determined by the node with the most work. In our experiments, we derive the cost model's parameters ($m, b, p$, and $t$) empirically using the database's performance on our heuristics-based physical planner.

```
tabuList = ∅ // list of all previous data-to-node assignments
Function TabuSearch
Input: S = {s_{i,j} i ∈ (1...n), j ∈ (1...k) // all slice sizes
Output: P {x_{i,j} ∀i, ∀j } //physical plan
P = minBandwidthHeuristic(S) // init to greedy placement
tabuList.insert(P); // current assignments cannot be repeated
PerNodeCosts = nodeCosts(P);
P' = ∅;
while P ≠ P' do
    P' = P; // store prev. version
    for all j ∈ (1...k) do
        if PerNodeCosts[j] ≥ meanNodeCost then
            P = RebalanceNode(j, P);
        end if
    end for
    PerNodeCosts = nodeCosts(P);
end while
return P;
```

```
Function RebalanceNode
Input: N, P // node for rebalancing, present join plan
Output: P' // revised plan
// for each join unit assigned to N
for all i | P[x_{i,N}] = 1 do
    // foreach candidate host
    for all j ∈ (1...k) do
        if j ≠ N ∧ ¬∃ tabuList(i,j) then
            w = P; w.assign(i,j);
            if queryCost(w) ≤ queryCost(P) then
                P = w; // new top plan
                tabuList.insert(i,j)
            end if
        end if
    end for
end for
return P;
```

**Algorithm 2:** Tabu search algorithm

## 5.2 Shuffle Planners

We now discuss the physical planner which takes slice statistics as input and produces an assignment of join units to nodes. We have designed a number of physical planners and each technique below optimizes the operator's data alignment cost, cell comparison cost, or both, and many use the cost model above.

**Minimum Bandwidth Heuristic** The network is the most scarce resource for joins in a shared-nothing architecture [24]. The Minimum Bandwidth Heuristic takes advantage of this observation by greedily assigning join units to nodes. For a given join unit, $i$, having slice sizes $\{s_{i,1}, \ldots, s_{i,k}\}$ over $k$ nodes, the physical planner identifies the node $n$ with the most cells from join unit $i$:

$$n = \arg\max_{j=[1,k]} s_{i,j} \tag{9}$$

and assigns the join unit to that node, i.e., $x_{i,n} = 1$. We call $n$ the join unit's "center of gravity" since it corresponds to the largest fraction of the join unit's cells. This heuristic provably minimizes the number of cells transmitted by a physical plan, but it does nothing to address unequal load in the cell comparison step.

**Tabu Search** We improve upon the minimum bandwidth incrementally using a variant of the Tabu Search [18]. This locally optimal search starts with the minimum bandwidth heuristic plan, and probes the physical plan space by attempting to move join units one at a time from nodes having a higher than average analytical cost to nodes with a lower cost. Once a join unit has been placed on a node once, it is added to a global "tabu" list, prohibiting the optimizer from re-evaluating this data-to-node assignment. The algorithm attempts to unburden the most costly nodes until it can no longer improve on the plan.

The Tabu pseudocode is shown in Algorithm 2. It begins by assigning slices to nodes based on their center of gravity to minimize their network transmissions. The algorithm then adds all of its greedy assignments to the tabu list. With this initial assignment, it calculates the join cost for each node by summing up their individual data alignment and cell comparison costs as defined in Equations 5-7. Here, instead of taking the $max$, the model considers a single $j$, or node value, at a time. It then adds all of the present assignments to the tabu list, to prevent reevaluation.

The algorithm then enters a while loop that iterates over cluster nodes, attempting to rebalance nodes with a greater than average analytical cost. The rebalancing operation iterates over every join unit assigned to an overburdened node. It executes a what-if analysis to see whether reassigning the join unit $j_i$ will reduce the entire plan's cost. Note that a reassignment is only valid if it does not appear on the tabu list. If the proposed change reduces the cost, a new plan is selected and the reassignment is recorded in the tabu list. This algorithm continues until the plan can no longer be improved.

We formulate the tabu list to cache the data-to-node assignments (or $x_{i,j}$'s that have ever been 1), rather than entire plans evaluated, for numerous reasons. First, network bandwidth is the most costly resource for this operator and the reassignment is a function of the local slice size, so moving join unit $i$ to node $j$ when it was there before and removed is unlikely to be profitable. Also, this approach makes maintaining an approximate list of previously evaluated plans much more efficient. Rather than recording all of the $x_{i,j}$ variables for each of the prior plans, Tabu only maintains a list of prior $x_{i,j}$ assignments. This makes probing the search space much more tractable, reducing it from exponential $2^{i \times j}$ to polynomial, $i \times j \propto n^2$. Lastly, this restriction prevents the planner from creating loops, wherein a join unit gets passed back and forth between two non-bottleneck nodes. This is particularly important when several nodes have high costs, as the optimizer only rebalances one node at a time. Hence, the most costly node is likely to change during a single round of rebalancing operations.

Tabu is attractive because it produces solutions that address both load balancing and data alignment without resorting to an exhaustive search of the query plan space. On the other hand, this planner may be susceptible to slowdown in the presence of many join units and nodes, which will complicate its planning.

**Integer Linear Program (ILP) Solver** Formulating the cost model in Section 5.1 as an integer linear program is a way to seek optimal physical shuffle join plans. This approach uses two structural variables $d$ and $g$, for the data alignment and cell comparison times, respectively. The ILP begins with the constraint in Equation 4, such that each join unit is assigned to exactly one node. The solver next models the costs of sending join unit slices by taking the complement of Equation 5. We use this approach because constraint solvers do not support conditional variables and cannot evaluate the negation in the cost model.

This constraint quantifies the highest send cost per node:

$$d - (-t \times \sum_{i=1}^{n} \sum_{j=1}^{k,j \neq d} x_{i,j} s_{i,d}) \geq 0 \tag{10}$$

The data receive time constraint for the node $d$, Equation 6, is:

$$d - (-t \times \sum_{i=1}^{n} x_{i,j}(S_i - s_{i,j})) \geq 0 \tag{11}$$

The integer program calculates the processing cost of each join unit based on its algorithm as:

$$C_i = \begin{cases} t_i \times b + u_i \times p & \text{if hash join} \\ m \times S_i & \text{if merge join} \end{cases}$$

The nested loop join is never profitable, as we demonstrate analytically in Section 4 and empirically in Section 6.1, hence we do not model it here. The solver also takes into account cell comparison time using the following constraint:

$$g - \sum_{i=1}^{n} x_{i,j} C_i \geq 0 \tag{12}$$

The ILP's objective function is: $min(g + m)$.

The structural variables $g$ and $m$ are used to subtract the cost of the join at each node. They have the added constraint that they must remain positive numbers, which in conjunction with the above constraints implements the $max$ functionality of the cost model.

We apply the SCIP integer program solver to find the optimal solution [2]. This approach is considered because it finds globally cost-effective solutions, whereas the Tabu search may become trapped at a local optimum. However, solvers also have difficulty scaling out to problems with thousands of decision variables.

**Coarse Solver** Empirically, we found that the ILP Solver has difficulty converging for shuffle join optimizations of moderate size (1024 join units). To address this, we assign join units at a coarser granularity. The Coarse Solver groups join units that share a center of gravity together to reduce the total number of assignments, and thus reduces the number of distinct decision variables. By evaluating slices that have their largest concentration of cells in the same location as a group, we make this problem easier to solve by ensuring that decision variables do not "conflict" with one another as they would if data were randomly packed. As an aside, costly "bin conflicts" occur in the solver when a bin has an equal concentration of cells on two or more hosts. This coarser technique is likely to make the solver run faster, however it may exact a cost in query plans of poorer quality by evaluating the join in larger segments.

# 6. EXPERIMENTAL RESULTS

We first describe our experimental setup and then we focus on our evaluation of the efficiency of our proposed framework.

**Experimental Setup** We conducted our physical planner and real-world data evaluations on a dedicated cluster with 4 nodes, each of which has 50 GB of RAM and is running Red Hat Enterprise Linux 6. This shared-nothing cluster transmitted data over a fully switched network. The remaining evaluations were done on a 12-host testbed, with each node having 2.1 GHz processors and 256 GB of RAM, all also using a switched network. In both settings, the database is stored on SATA disks. Each experiment started with a cold cache and all were executed 3 times. We report the average query duration.

## 6.1 Logical Planning Evaluation

We begin by verifying the logical query planner in Section 4. Here, it is our goal is a) to demonstrate that we identify the best execution plan and b) to validate the planner's cost model, showing that the optimizer can accurately compare competing query plans. Our experiment uses two 64 MB synthetic arrays, sized to make the nested loop join, which has a polynomial runtime, a feasible option. The two input arrays have the schema: `A<v:int>[i=1,128M,4M]` and `B<w:int>[j=1,128M,4M]` and we execute the A:A query: `SELECT * INTO C<i:int, j:int>[v=1,128M,4M]; FROM A, B WHERE A.v = B.w`. Here, we use a single node to test our logical planner while controlling for physical planning.

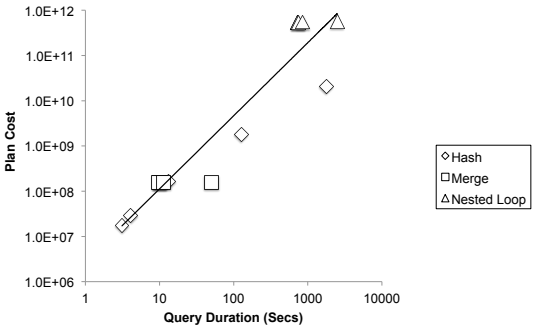


Figure 5: Logical plan cost vs. query latency.

This query has a wide variety of plausible plans; all of its attributes may become dimensions and vice versa and there are several join algorithms to choose from. Figure 5 and 6 compare three plans. *Merge* refers to the plan `mergeJoin(redim(A,C), redim(B,C))`. *Hash* refers to `redim(hashJoin(hash(A), hash(B)),C)`. The *Nested Loop* plan is similar to Hash with the only change being the join algorithm.

Because the cost model of the logical planner works at the level of entire arrays, its decisions are based on the output cardinality of the join. To capture a large cross-section of potential plans, we evaluated all three join algorithms, varying the selectivity of the join. Hence, if $A$ and $B$ have $n_a$ and $n_b$ cells respectively, a join with selectivity 0.1 produces $0.1 \times (n_a + n_b)$ output cells. We evaluated the join with selectivity of 0.01, 0.1, 1, 10, and 100 to capture a wide range of outcomes.

Figure 5 plots the duration of each query against its logical plan cost.The results revealed a strong power law correlation between query duration and projected cost ($r^2 = 0.9$), indicating that the planner will be able to accurately compare its options. In fact, for all 5 selectivities, the plan with the minimum cost also had the shortest duration. This study also verified that the nested loop join is never a profitable plan. This makes sense in light of its polynomial runtime.

Figure 6 displays the time of each of the logical plans over different join selectivities. All of these performance curves see a significant rise in latency as their output cardinality grows. This effect is caused by the overhead of managing larger output chunks, including allocating more memory and having less locality in the CPU cache. Hash and nested loop joins are especially sensitive to high match rates in the join because they also sort the data after cell comparison. All join deviates from the trend when the data produces an output 100 times larger than its sources. At first glance, it might seem that we should account for this delay in the model, but this is not necessary because all possible plans produce the same output and thus bear this cost.

This result also demonstrates that for queries with a low selectivity, the hash join is fastest. Because the hash join operates over unordered buckets, the plan cannot sort the query data until after the cell comparison phase, and this expensive step operates on fewer cells. Once we reach a selectivity of 1, where the join's inputs and outputs have the same size, merge join narrowly edges out the hash join plan. Merge join executes redimensions on the two source arrays, performing twice as many sorts each on half the number of cells. In other words, this logarithmic time operation scales better in smaller batches. As the join produces higher output cardinalities, merge join becomes the clear best choice because it front loads the the reordering needed for the destination schema. For the largest output cardinality, merge join performs 35X faster than hash join.

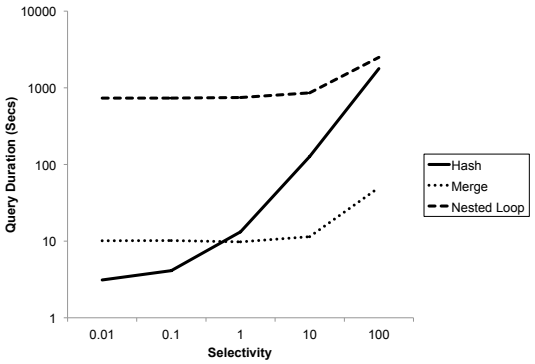


Figure 6: Performance for different logical plans and selectivities.

These results demonstrate how crucial it is to pick the right logical plan for a join query. If the query calls for reordering the data, correctly placing the sort before or after cell comparison makes a dramatic difference in the database's performance.

## 6.2 Physical Planning Evaluation

We now consider the performance of the physical join planners. This section begins by verifying analytical cost model. We then examine the performance of the optimizers for merge and hash joins under a variety of data distributions. Since we established that the nested loop join has the worst performance above, it is excluded from this study.

The joins are evaluated by joining two 100 GB arrays with schema: `A<v1:int, v2:int>[i=1,64M,2M, j=1,64M,2M]`. Their array space is a grid with 32x32 chunks, hence the merge join has 1024 join units to optimize. Likewise, the hash join experiments assign 1024 hash buckets to its nodes.

Our experiments use synthetic data to carefully control the level of skew in the source arrays. They begin with uniformly distributed data, wherein all of the slices are the same size, and gradually increase the skew. Here, the join unit and slice sizes follow a Zipfian distribution. This distribution's skewness is characterized with $\alpha$, and higher $\alpha$'s denote greater imbalance in the data sizes. For these trials we have a very low selectivity of 0.0001; this design tests extreme differences in size between the two sides of a join unit, since if a massive chunk is being compared with one that has very few cells there are limited opportunities to match cells.

**Baseline** We compare the shuffle join planners to a naive planner. This baseline is not skew-aware, and it makes decisions at the level of entire arrays. For merge joins, this approach simply moves the smaller array to the larger one. For hash joins, the planner assigns an equal number of buckets to each node. If the join has $b$ hash buckets, and $k$ nodes, the first $\lceil b/k \rceil$ buckets are assigned to the first node and each subsequent host is assigned the next same-sized slice of buckets. We take this approach from relational optimizers [31].

**ILP Solvers** Both the ILP Solver and Coarse ILP Solver have a time budget within which to complete their optimization. This limit is a workload-specific parameter, tuned to an empirically observed time at which the solver's solution quality becomes asymptotic. We did this to make the approach competitive with the alternatives, each of which completes its planning phase, assigning join units to nodes, in few seconds. Our experiments have a time budget of 5 minutes. The Coarse ILP Solver packs its join units into 75 bins.

Our experiments display the time of shuffle join planning (*Query Plan*), data alignment (*Data Align*), and cell comparison (*Cell Comp*) for the Baseline (*B*), ILP Solver (*ILP*), ILP-Coarse Solver (*ILP-C*), Minimum Bandwidth Heuristic (*MBH*) and Tabu Search (*Tabu*) physical planners.

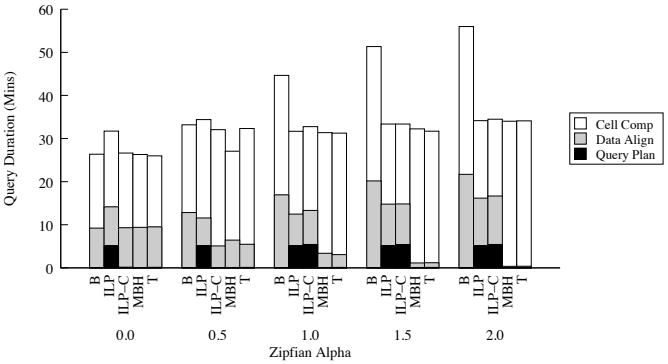


Figure 7: Merge join with varying skew and physical planners.

| | ILP | | ILP-Coarse | | Tabu | |
|---|---|---|---|---|---|---|
| Skew | Time | Cost | Time | Cost | Time | Cost |
| $\alpha = 1.0$ | 33.5 | 221 | 37.9 | 236 | 39.9 | 259 |
| $\alpha = 1.5$ | 20.0 | 131 | 22.9 | 143 | 23.5 | 151 |
| $\alpha = 2.0$ | 10.9 | 69 | 11.5 | 75 | 11.6 | 75 |

Table 2: Analytical cost model vs. join time in minutes for hash join.

**Analytical Model Verification** We demonstrate the efficacy of the cost model from Section 5.1 in Table 2. The table shows the model's qualitative estimates in comparison to the summed data alignment and join execution times for the cost-based physical planners. The results are taken from the hash joins shown in Figure 8 under moderate-to-high skew. We chose this series because it demonstrates a complex optimization scenario with many slices per join unit. These queries also showcase pronounced processing skew, for which the physical planners are optimized.

A linear model shows that plan costs are very well-correlated with the observed query latency ($r^2$ = 0.9). This implies that the planners leveraging this model are able to accurately compare competing plans. We see a minor outlier where $\alpha$=2. ILP Coarse and Tabu produce solutions with the same cost, but Tabu's plan runs for 6 seconds longer. For queries that execute for tens of minutes apiece, we posit that this is an acceptable level of variance.

### 6.2.1 Merge Join

We now test the efficiency of the physical planners for managing skew in merge joins. This experiment evaluates the D:D query:

```
SELECT  A.v1 - B.v1, A.v2 - B.v2
FROM A, B
WHERE A.i = B.i AND A.j = B.j;
```

by executing the plan `merge(A,B)`; its slices are whole chunks. The query runtimes are shown in Figure 7 for different physical planner and skew levels. For the uniform test, $\alpha = 0$, all optimizers produce plans of similar quality. The ILP Solver performs noticeably slowly, because it attempts to make a series of small, inconsequential improvements over the uniformly distributed data and cannot converge on a globally optimal plan.

As skew increases, the planners all adapt to exploit this change. The ILP Solver time decreases as the difference between the best plan and the alternatives becomes more pronounced. Tabu converges quickly as it too identifies areas of beneficial skew. Generally speaking, because this join plans at the level of logical chunks, the optimizer has at most two choices for where it can cost-effectively assign each join unit: the location of its two inputs. All of the alternatives incur additional time in network bandwidth, and this is often the bottleneck for array joins.

Owing to this simplicity in the plan search space, the Minimum Bandwidth Heuristic performs best. Its planning time is virtually nonexistent and bringing sparse chunks to their denser counterparts is enough to minimize query durations in this context. This

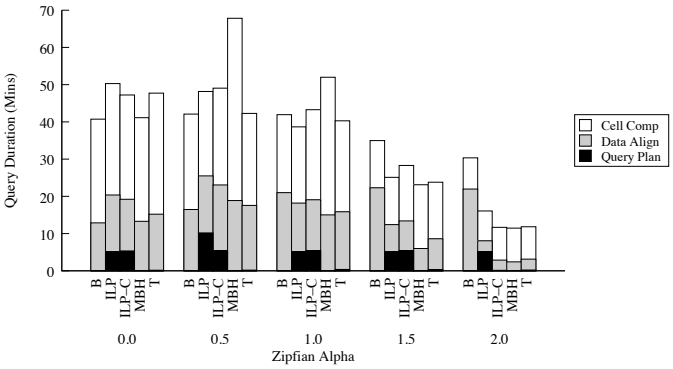


Figure 8: Hash join with varying skew and physical planners.

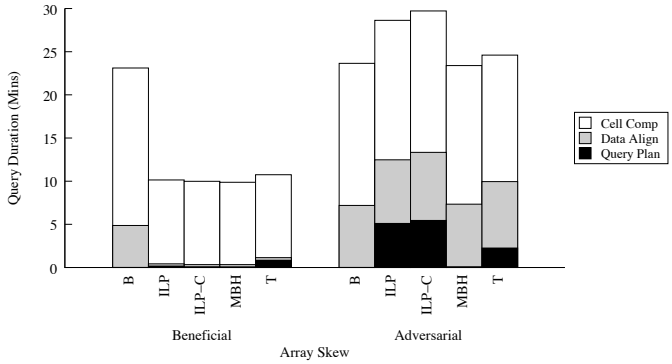


Figure 9: Merge join using real world data, organized by join planner.

approach also benefits from having an uncomplicated cost mechanism. By not individually calculating the cost of cell comparison and data alignment on a per-cell basis, it is robust to hardware-level variance. For example, cells from the data alignment step may already be present in memory at cell comparison time, but locally stored source data needs to be fetched from disk, creating differences in what is analytically regarded as the same operation.

### 6.2.2 *Hash Join*

We now examine the performance of the physical planners for hash joins. This experiment evaluates the A:A query:

```
SELECT  A.i, A.j, B.i, B.j
INTO<A.i:int, A.j:int, B.i:int, B.j:int>[]
FROM A, B
WHERE A.v1 = B.v1 and A.v2 = B.v2;
```

under different levels of skew; its join units are hash buckets. The findings in Figure 8 introduce skew both in the join unit sizes and their distribution across nodes. Each join unit is spread over all nodes, creating a more complicated search space for the planners.

For uniformly distributed data, the planners all evenly distribute join execution over the cluster. The ILP solvers suffer from longer execution times because they rarely find the optimal solution at a speed that would make them competitive. In contrast, the MBH produces the most cost effective plan. It reduces the time of data alignment by moving the smallest side of each join unit. Because all of the join units are of approximately the same size, each node is assigned roughly the same amount of work.

On the other hand, MBH performs exceptionally poorly under slight skew, where $\alpha = 0.5$. Here, its locally optimal, single-pass solution latches on to small differences in slice size and creates significant imbalance during the cell comparison step. This issue is compounded by low skew that forces the hash join to create large hash maps. Building this data structure is substantially more costly than probing it. As the skew increases, the hash join selects the smaller side of each join unit as its build array, reducing this effect.

The ILP Solver also performed poorly in the presence of slight skew. For $\alpha = 0.5$, the planner failed to produce a valid plan within its allotted time budget. To accommodate this limitation, we evaluated this query with a one-time 10 minute window. Even so, it's solution was less effective than the one provided by Tabu.

Overall, the results demonstrate that Tabu performs the best for hash joins. This algorithm starts out with the greedy heuristic to reduce data alignment time and progresses on to a locally optimal search of the plan space to balance the cell comparison load. Hence, it capitalizes on the best elements of heuristics and sophisticated modeling to make this complicated join's end-to-end latency low.

## 6.3 Real-World Data Evaluation

The next set of experiments concern two datasets sourced from real-world science applications. The first consists of 170 GB of satellite imagery recorded by the NASA MODIS platform [25] over a one week period. Scientists use MODIS data to model sea, atmosphere, and land processes. MODIS records light measurements at various wavelengths (or "bands") accompanied by uncertainty information. Its data is relatively evenly distributed over array space.

The second dataset, AIS, is a ship tracking database for commercial vessels provided by the National Oceanographic and Atmospheric Administration [26]. This dataset contains one year of marine traffic with a stored size of 110 GB. NOAA collects location broadcasts from ships, and the array's attributes include the track's ship identifier, course, speed, and rate of turn. This dataset exhibits high amounts of skew, as vessels often cluster around major ports and traverse shipping lanes. Its arrays are span the coastal waters of the United States. Both sources have three dimensions: time, longitude, and latitude. Their latitude-longitude dimensions are divided into 4° x 4° chunks, producing in 4,186 join units.

We use these two datasets to test the physical planners on a merge join. All of the joins in this section have sorted chunks as their join units. The experiments first examine beneficial skew by joining AIS with MODIS data, and then adversarial distributions via querying two MODIS bands.

### 6.3.1 *Beneficial Skew*

Recall that MODIS data is used to model the earth's atmosphere, and in this experiment we study the satellite measurements taken at the location of ship tracks to understand the environmental impact of marine traffic. This experiment executes the query:

```
SELECT Band1.reflectance,Broadcast.ship_id
FROM Band1, Broadcast
WHERE Band1.longitude = Broadcast.longitude
  AND Band1.latitude = Broadcast.latitude;
```

It joins on the geospatial dimensions alone to produce a long-term view of the environment. This query is an example of beneficial skew. For AIS, nearly 85% of the data is stored in just 5% of the chunks. In contrast, MODIS has only slight skew; the top 5% of its chunks contain only 10% of the data. Hence, although MODIS data is uniformly distributed over its array space, the AIS data has several hotspots along the US coastline. This distribution is beneficial because the AIS chunk sizes are very polarized, hence there is always a clear winner as to which assignment will reduce the bandwidth used during data alignment.

Figure 9 shows that the shuffle join planners achieve a speedup of nearly 2.5X over the baseline in end-to-end performance. This confirms our findings in Section 6.2; under severe skew, cost-based planning makes a big difference in performance. Data alignment is reduced by almost 20X, as the planners exploit beneficial skew by moving sparse slices. Cell comparison is also halved compared to the baseline, as its per-node data distribution is simply more even. The baseline moved the smaller, more skewed AIS array to align it

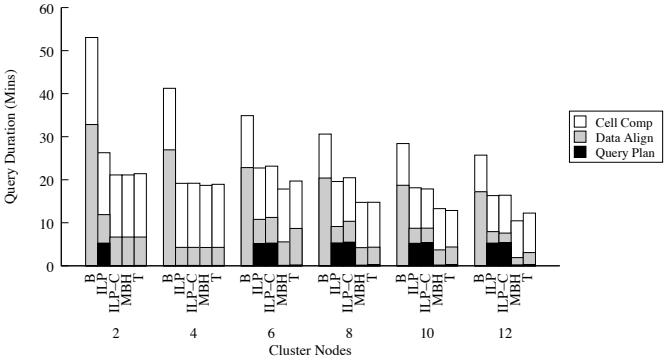

Figure 10: Scale out performance of merge join

with the more uniform MODIS source cells, transferring many of its largest chunks to the same node. Thus the skew-agnostic planner increased the duration of both join steps.

### 6.3.2 *Adversarial Skew*

Our experiment on this dataset joins two MODIS bands, calculating the normalized difference vegetation index as in [28]. The query's source arrays have very slight skew, as chunks nearer to the equator having more cells. This is an artifact of latitude-longitude space being more sparse near the poles of the globe.

```
SELECT (Band2.reflectance - Band1.reflectance)
     / (Band2.reflectance + Band1.reflectance)
FROM Band1, Band2
WHERE Band1.time = Band2.time
  AND Band1.longitude = Band2.longitude
  AND Band1.latitude = Band2.latitude;
```

The join's output consists of a 3D array with a single attribute for the vegetation reading, and this query is an example of adversarial skew. Because both bands collect their data from the same sensor array, their resulting chunks for a given location are very close in size. On average the difference between joining chunks is 10,000 cells, whereas the mean chunk size is 665,000 cells.

All of the planners produce queries with comparable execution times. This demonstrates that the optimizers can achieve significant speedups in the presence of skew without a commensurate loss in performance for uniformly distributed data. That being said, the shuffle algorithms that search for a plan, i.e., Tabu and the solvers, incur overhead as they scale up to a larger number of chunks. They do so because each incremental plan revision can only result in small changes in plan cost, complicating their decision space. Future work could address this challenge by approximating the chunk sizes in the planners for arrays having many chunks, in effect, rounding the chunk sizes by some increment. This would make it such that join units that are almost the same size are not considered distinct options to the optimizers.

## 6.4 Scale Out Test

We now test the shuffle join's ability to scale out to many nodes. This experiment executes the merge join in Section 6.2.1 using a range of cluster sizes. The database runs on 2-12 nodes in even numbered quantities. Here, the level of skew is held constant at a $\alpha = 1.0$. The resulting query latencies are displayed in Figure 10.

The skew-aware planners chose plans that execute faster with two nodes than the baseline plan with 12 nodes. With very few nodes, the join spends most of its time aligning data. In the two-node trial, the hosts can only send or receive from a single network link, and this limited level of parallelism exacerbates the network bottleneck. On the other hand, this setting has much faster optimization times for the cost-based planners. The ILP solvers quickly converge on the optimal solution in the smallest scale trial.

As more nodes are introduced, the physical planning takes on an exponentially richer decision space and the ILP optimizers use their entire time budget. As they have more chunk-to-node assignment options, their plans are not high-quality enough to justify this wait time. Thus the much simpler Minimum Bandwidth Heuristic performs best overall. At the small scale configuration, MBH performs on par with the more sophisticated approaches and as the cluster grows it surpasses them, cutting execution time in half.

# 7. RELATED WORK

Parallel, shared-nothing databases were proposed in [35], implemented in [8, 11, 15], and surveyed in [12].The requirements for array data management were researched in [3, 19, 36]. Several efforts are underway to implement scientific databases [5, 6, 9], but none of these optimize join processing over skewed data. Research in parallel join execution for the relational model has focused on the performance of hash joins [13, 21]. These algorithms are evaluated in [31]. We build upon these results, but decouple the data alignment of join inputs from their cell comparison.

Modeling skew for parallel joins has been researched for relational databases only. In [4, 7] they showed that skewed data distributions offer performance improvement opportunities for in-memory hash-joins. [38] explored the impact of skew on query execution plans. There are also several skew-aware optimizations for parallel hash joins, including bucket size tuning [20], recursive hash bucket splitting [41], and global tuple-level load balancing [14]. [40] addresses load balancing for sort-merge joins. Each of these approaches optimizes for execution skew and we also consider this objective in our models. Array workloads, however, are much more sensitive to network and I/O availability, and we created a series of techniques to address this. Finally, minimizing the network transmission cost and balancing the cpu cost has been extensively studied for relational joins in [17, 23, 39].

The performance benefits gained through exploiting data locality specifically for parallel, in-memory hash joins were explored for relational data [30, 29]. Our work also takes into account the location of data when planning a join, however we do so for a wider variety of join implementations within an array database where sort order is fundamental to its design.

# 8. CONCLUSIONS

We introduce the *shuffle join optimization* framework that offers efficient join execution plans in the presence of skew for array databases. It includes a logical optimizer that exploits the spatial storage and sorting properties of the input data to generate array-level join execution plans that minimize the hardware resources needed for a join. The framework also has a physical planner for executing these operators across the database cluster by assigning non-overlapping array subsets to nodes. An analytical cost model enables the physical planner to exploit skew to craft efficient data-to-node assignments. Experimental results show that this tactic achieved a 2.5X speedup for skewed data and performed comparably to skew-agnostic techniques for uniform distributions.

There are several promising directions for future work from this study. Identifying the most efficient order of several joins within a single query is one such question. Another area warranting further investigation is generalizing this two-step optimization model to complex analytics that combine arrays, such as covariance matrix queries.

# 9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This research was funded by the Intel Science and Technology Center for Big Data and by NSF IIS 1253196.

## 10. REFERENCES


[1] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, 2005.

[2] T. Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[3] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Communications of the ACM*, 53(6):68–78, 2010.

[4] C. Balkesen, J. Teubner, G. Alonso, and T. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying Hardware. In *ICDE*, 2013.

[5] A. Ballegooij. Ram: A multidimensional array dbms. In *EDBT Workshops '05*.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *SIGMOD Record*, 1998.

[7] S. Blanas, Y. Li, and J. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, 2011.

[8] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):4–24, 1990.

[9] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[10] P. Cudré-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. A Demonstration of SciDB: A Science-Oriented DBMS. *PVLDB*, 2(2):1534–1537, 2009.

[11] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *Trans. on Knowledge & Data Eng.*, 2(1):44–62, 1990.

[12] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, 1990.

[13] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*, volume 14. June 1984.

[14] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, volume 92, pages 27–40, 1992.

[15] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of non-stop sql release 2 demonstrating near-linear speedup and scaleup on large databases. *Tandem Tech Report*, 1989.

[16] C. Faloutsos, B. Seeger, A. Traina, and C. Traina, Jr. Spatial join selectivity using power laws. *SIGMOD Rec.*, 29(2), May 2000.

[17] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *SIGMOD*, 1992.

[18] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[19] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[20] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer. In *VLDB*, 1990.

[21] 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.

[22] Large Synoptic Survey Telescope. `http://www.lsst.org`.

[23] J. Li, A. Deshpande, and S. Khuller. Minimizing Communication Cost in Distributed Multi-query Processing. In *ICDE*, 2009.

[24] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[25] NASA, MODIS Website. `modis.gsfc.nasa.gov/data/`.

[26] National Oceanic and Atmospheric Administration. *Marine Cadastre*. `http://marinecadastre.gov/AIS/`.

[27] M. E. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[28] G. Planthaber, M. Stonebraker, and J. Frew. Earthdb: scalable analysis of modis data using scidb. In *BigSpatial '12*.

[29] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. SIGMOD '14, pages 1483–1494, New York, NY, USA, 2014. ACM.

[30] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.

[31] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD '89*.

[32] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[33] Sloan Digital Sky Survey. `http://www.sdss.org`.

[34] E. Soroush, M. Balazinska, and D. L. Wang. ArrayStore: a storage manager for complex parallel array processing. In *SIGMOD*, 2011.

[35] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 1986.

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

[37] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: a complex analytics genomics benchmark. In *SIGMOD Conference*, pages 177–188, 2014.

[38] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB*, 1991.

[39] X. Wang, R. Burns, A. Terzis, and A. Deshpande. Network - Aware Join Processing in Global Scale Database Federations. In *ICDE*, 2008.

[40] J. Wolf, D. Dias, and P. Yu. A parallel sort merge join algorithm for managing data skew. *Trans. on Parallel and Dist. Systems*, 1993.

[41] J. Wolf, P. Yu, J. Turek, and D. Dias. A parallel hash join algorithm for managing data skew. *Trans. on Parallel and Dist. Systems*, 1993.