# Cost Models for Query Processing Strategies in the Active Data Repository *

Chialin Chang
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland, College Park 20742

chialin@cs.umd.edu

**Abstract**

Exploring and analyzing large volumes of data plays an increasingly important role in many domains of scientific research. We have been developing the Active Data Repository (ADR), an infrastructure that integrates storage, retrieval, and processing of large multi-dimensional scientific datasets on distributed memory parallel machines with multiple disks attached to each node. In earlier work, we proposed three strategies for processing range queries within the ADR framework. Our experimental results show that the relative performance of the strategies changes under varying application characteristics and machine configurations. In this work we describe analytical models to predict the average computation, I/O and communication operation counts of the strategies when input data elements are uniformly distributed in the attribute space of the output dataset, restricting the output dataset to be a regular $d$-dimensional array. We validate these models for various synthetic datasets and for several driving applications.

## 1 Introduction

The exploration and analysis of large datasets is playing an increasingly central role in many areas of scientific research. Over the past several years we have been actively working on data intensive applications that employ large-scale scientific datasets, including applications that explore, compare, and visualize results generated by large scale simulations [8], visualize and generate data products from global coverage satellite data [4], and visualize and analyze digitized microscopy images [1]. Such applications often use only a subset of all the data available in both the input and output datasets. References to data items are described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset(s). Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved and processed. The processing structures of these applications also share common characteristics. Figure 1 shows high-level pseudo-code for the basic processing loop in these applications. The processing steps consist of retrieving input and output data items that intersect the range query (steps 1–2 and 4–5), mapping the coordinates of the retrieved input items to the corresponding output items (step 6), and aggregating, in some way, all the retrieved input items mapped to the same output data items (steps 7–8). Correctness of the output data values usually does not depend on the order input data items are aggregated. The mapping function, $Map(i_e)$, maps an input item to a set of output items. We extend the computational model to allow for an intermediate data structure, referred to as an *accumulator*, that can

*This research was supported by the National Science Foundation under Grant #ACI-9619020 (UC Subcontract # 10152408) and the Office of Naval Research under Grant #N6600197C8534.

$O \leftarrow$ Output Dataset, $I \leftarrow$ Input Dataset
(* Initialization *)
1. **foreach** $o_e$ **in** $O$ **do**
2.   **read** $o_e$
3.   $a_e \leftarrow Initialize(o_e)$
(* Reduction *)
4. **foreach** $i_e$ **in** $I$ **do**
5.   **read** $i_e$
6.   $S_A \leftarrow Map(i_e)$
7.   **foreach** $a_e$ **in** $S_A$ **do**
8.     $a_e \leftarrow Aggregate(i_e, a_e)$
(* Output *)
9. **foreach** $a_e$ **do**
10.   $o_e \leftarrow Output(a_e)$
11.   **write** $o_e$

Figure 1: The basic processing loop in the target applications.

be used to hold intermediate results during processing. For example, an accumulator can be used to keep a running sum for an averaging operation. The aggregation function, $Aggregate(i_e, a_e)$, aggregates the value of an input item with the intermediate result stored in the accumulator element ($a_e$). The output dataset from a query is usually much smaller than the input dataset, hence steps 4–8 are called the *reduction* phase of the processing. Accumulator elements are allocated and initialized (step 3) before the reduction phase. The intermediate results stored in the accumulator are post-processed to produce final results (steps 9–11).

We have been developing the Active Data Repository (ADR) [2], a software system that efficiently supports the processing loop shown in Figure 1, integrating storage, retrieval, and processing of large multi-dimensional scientific datasets on distributed memory parallel machines with multiple disks attached to each node. ADR is designed as a set of modular services implemented in C++. Through use of these services, ADR allows customization for application specific processing (i.e. the $Initialize$, $Map$, $Aggregate$, and $Output$ functions described above), while providing support for common operations such as memory management, data retrieval, and scheduling of processing across a parallel machine. The system architecture of ADR consists of a front-end and a parallel back-end. The front-end interacts with clients, and forwards range queries with references to user-defined processing functions to the parallel back-end. During query execution, back-end nodes retrieve input data and perform user-defined operations over the data items retrieved to generate the output products. Output products can be returned from the back-end nodes to the requesting client, or stored in ADR.

In earlier work [3, 7], we described three potential processing strategies, and evaluated the relative performance of these strategies for several application scenarios and machine configurations. Our experimental results showed that the relative performance of the strategies changes under varying application characteristics and machine configurations. In this paper we describe analytical models to predict the average operation counts of the strategies when input data elements are uniformly distributed in the attribute space of the output dataset, restricting the output dataset to be a regular $d$-dimensional array. We validate these cost models with queries for synthetic datasets and for several driving applications [1, 4, 8].

## 2 Overview of ADR

In this section we briefly describe three strategies for processing range queries in ADR. First we briefly describe how datasets are stored in ADR, and outline the main phases of query execution in ADR. More detailed descriptions of these strategies and of ADR in general can be found in [2, 3, 7].

### 2.1 Storing Datasets in ADR

A dataset is partitioned into a set of chunks to achieve high bandwidth data retrieval. A chunk consists of one or more data items, and is the unit of I/O and communication in ADR. That is, a chunk is always retrieved, communicated and computed on as a whole during query processing. Every data item is associated with a point in a multi-dimensional attribute space, so every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates (in the associated attribute space) of all the data items in the chunk. In the remaining of this paper, we use the MBR of a chunk to determine the extent, the volume, the mid-point, and the top-right corner of the chunk. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multi-dimensional space placed in the same chunk. Chunks are distributed across the disks attached to ADR back-end nodes using a declustering algorithm [5, 9] to achieve I/O parallelism during query processing. Each chunk is assigned to a single disk, and is read and/or written during query processing only by the local processor to which the disk is attached. If a chunk is required for processing by one or more remote processors, it is sent to those processors by the local processor via interprocessor communication. After all data chunks are stored into the desired locations in the disk farm, an index (e.g., an R-tree [6]) is constructed using the MBRs of the chunks. The index is used by the back-end nodes to find the local chunks with MBRs that intersect the range query.

### 2.2 Query Processing in ADR

Processing of a query in ADR is accomplished in two steps; query planning and query execution.

A plan specifies how parts of the final output are computed and the order the input data chunks are retrieved for processing. Planning is carried out in two steps; *tiling* and *workload partitioning*. In the tiling step, if the output dataset is too large to fit entirely into the memory, it is partitioned into *output tiles*. Each output tile contains a distinct subset of the output chunks, so that the total size of the chunks in an output tile is less than the amount of memory available for output data. Tiling of the output implicitly results in a tiling of the input dataset. Each input tile contains the input chunks that map to the output chunks in the output tile. Similar to data chunks, an output tile is associated with a MBR that encompasses the MBRs (in the associated attribute space) of all the output chunks in the tile. During query processing, each output tile is cached in main memory, and input chunks from the required input tile are retrieved. Since a mapping function may map an input element to multiple output elements, an input chunk may appear in more than one input tile if the corresponding output chunks are assigned to different tiles. Hence, an input chunk may be retrieved multiple times during execution of the processing loop. In the workload partitioning step, the workload associated with each tile (i.e. aggregation of input items into accumulator chunks) is partitioned across processors. This is accomplished by assigning each processor the responsibility for processing a subset of the input and/or accumulator chunks.

The execution of a query on a back-end processor progresses through four phases for each tile:

1. **Initialization**. Accumulator chunks in the current tile are allocated space in memory and initialized. If an existing output dataset is required to initialize accumulator elements, an output chunk is retrieved by the processor that has the chunk on its local disk, and the chunk is forwarded to the processors that require it.

2. **Local Reduction**. Input data chunks on the local disks of each back-end node are retrieved and aggregated into the accumulator chunks allocated in each processor's memory in phase 1.
3. **Global Combine**. If necessary, results computed in each processor in phase 2 are combined across all processors to compute final results for the accumulator chunks.
4. **Output Handling**. The final output chunks for the current tile are computed from the corresponding accumulator chunks computed in phase 3.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output dataset has been computed. To reduce query execution time, ADR overlaps disk operations, network operations and processing as much as possible during query processing. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the queues are polled and, upon their completion, new asynchronous operations are initiated when there is more work to be done and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion.

## 2.3 Query Processing Strategies

In the following discussion, we refer to an input/output data chunk stored on one of the disks attached to a processor as a *local* chunk on that processor. Otherwise, it is a *remote* chunk. A processor *owns* an input or output chunk if it is a local input or output chunk. A *ghost chunk* is a copy of an accumulator chunk allocated in the memory of a processor that does not own the corresponding output chunk.

In the tiling phase of all the strategies described in this section, we use a *Hilbert space-filling curve* [5] to create the tiles. The goal is to minimize the total length of the boundaries of the tiles, by assigning chunks that are spatially close in the multi-dimensional attribute space to the same tile, to reduce the number of input chunks crossing tile boundaries. The advantage of using Hilbert curves is that they have good clustering properties [9], since they preserve locality. In our implementation, the mid-point of the bounding box of each output chunk is used to generate a Hilbert curve index. The chunks are sorted with respect to this index, and selected in this order for tiling.

**Fully Replicated Accumulator (FRA) Strategy.** In this scheme each processor performs processing associated with its local input chunks. The output chunks are partitioned into tiles, each of which fits into the available local memory of a single back-end processor. When an output chunk is assigned to a tile, the corresponding accumulator chunk is put into the set of local accumulator chunks in the processor that owns the output chunk, and is assigned as a ghost chunk on all other processors. This scheme effectively replicates all of the accumulator chunks in a tile on each processor, and during the *local reduction* phase, each processor generates partial results for the accumulator chunks using only its local input chunks. Ghost chunks with partial results are then forwarded to the processors that own the corresponding output (accumulator) chunks during the *global combine* phase to produce the complete intermediate result, and eventually the final output product.

**Sparsely Replicated Accumulator (SRA) Strategy.** The FRA strategy replicates each accumulator chunk in every processor, even if no input chunks will be aggregated into the accumulator chunks in some processors. This results in unnecessary initialization overhead in the *initialization* phase of query execution, and extra communication and computation in the *global combine* phase. The available memory in the system also is not efficiently employed, because of unnecessary replication. Such replication may result in more tiles being created than necessary, which may cause a large number of input chunks to be retrieved from

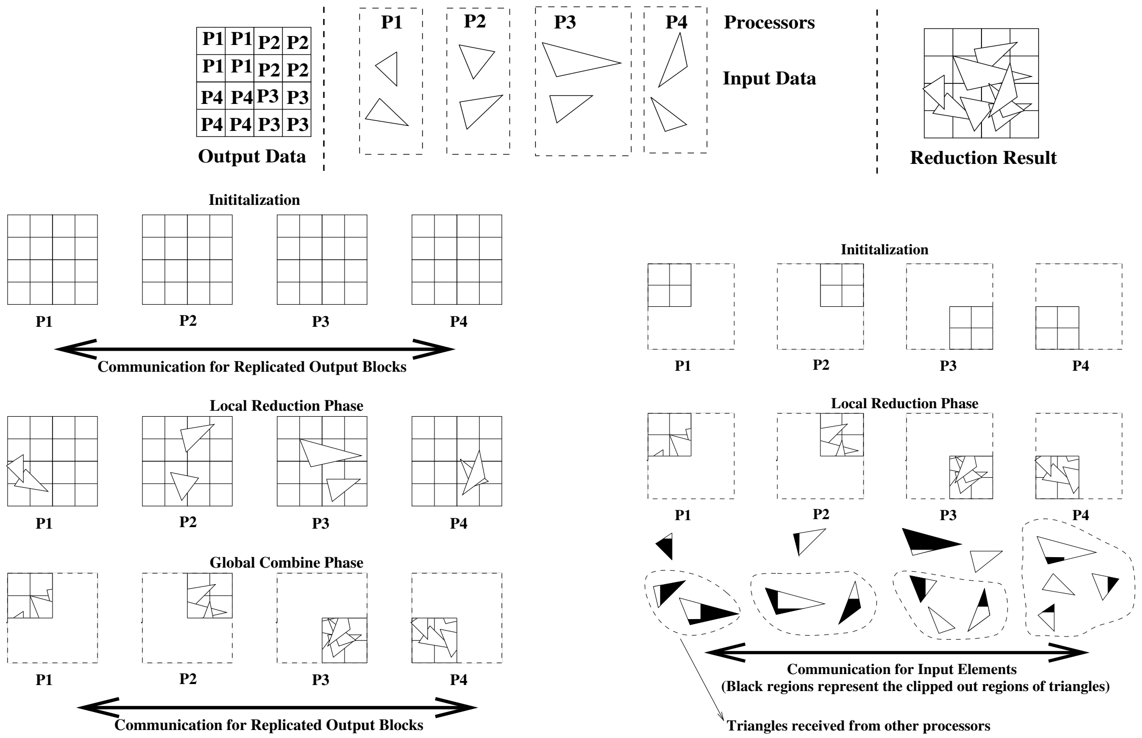



Figure 2: FRA strategy (left) and DA strategy (right).

disk more than once. In SRA strategy, a ghost chunk is allocated only on processors owning at least one input chunk that maps to the corresponding accumulator chunk.

**Distributed Accumulator (DA) Strategy.** In this scheme, every processor is responsible for all processing associated with its local output chunks. Tiling is done by selecting, for each processor, local output chunks from that processor until the memory space allocated for the corresponding accumulator chunks in the processor is filled. As in the other schemes, output chunks are selected in Hilbert curve order.

Since no accumulator chunks are replicated by the DA strategy, no ghost chunks are allocated. This allows DA to make more effective use of memory and produce fewer tiles than the other two schemes. As a result, fewer input chunks are likely to be retrieved for multiple tiles. Furthermore, DA avoids interprocessor communication for accumulator chunks during the *initialization* phase and for ghost chunks during the *global combine* phase, and also requires no computation in the *global combine* phase. On the other hand, it introduces communication in the *local reduction* phase for input chunks; all the remote input chunks that map to the same output chunk must be forwarded to the processor that owns the output chunk. Since a projection function may map an input chunk to multiple output chunks, an input chunk may be forwarded to multiple processors.

Figure 2 illustrates the FRA and DA strategies for an example application. One possible distribution of input and output chunks to the processors is illustrated at the top. Input chunks are denoted by triangles while output chunks are denoted by rectangles. The final result to be computed by reduction (aggregation) operations is also shown.

## 3 Assumptions and Definitions

In this section, we describe assumptions about our cost models and the definitions of parameters to be used in the models. We also define how the MBR of an output tile is partitioned into regions for analysis purpose. As to be seen later, the regions will be used to estimate the expected number of output tiles an input chunk maps to and the expected number of messages that DA generates for an input chunk during the local reduction phase.

### 3.1 Assumptions

The cost models presented in this paper make the following assumptions.

- A shared-nothing architecture with local disks is employed.
- All processors are assumed to have the same amount of memory.
- All input chunks are of the same number of bytes, and have the same extent when mapped into the output attribute space.
- All output chunks are of the same number of bytes, and their extents form a regular multi-dimensional grid.
- All input chunks map to the same number of output chunks, and all output chunks are mapped to by the same number of input chunks.
- An accumulator chunk is assumed to have the same number of bytes as that of its corresponding output chunk.
- Input chunks and output chunks are assumed to be distributed among processors by a declustering algorithm that achieves perfect declustering; that is, all the input (output) chunks whose MBRs intersect a given range query are distributed to as many processors as possible.

### 3.2 Definitions

We define the parameters to be used in the rest of this paper.

$I$ : the total number of input chunks required by the given query.

$U$ : the total number of output chunks to be computed by the given query.

$i$ : the size of an input chunk.

$u$ : the size of an output chunk.

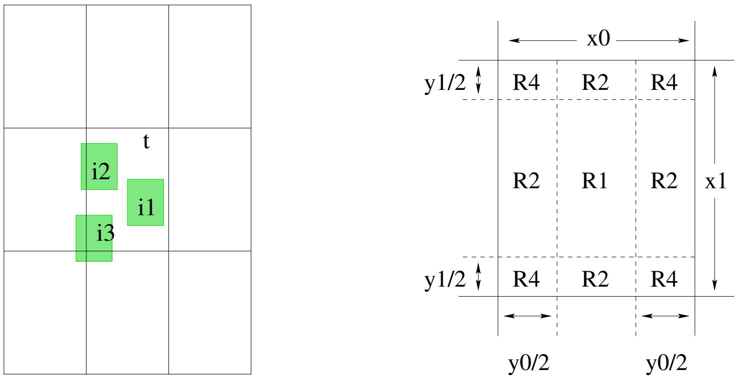
$P$ : the number of processors.

$M$ : the available memory size of a processor.

$\alpha$ : the number of output chunks an input chunk maps to.

$\beta$ : the number of input chunks an output chunk is mapped to.

$d$ : the number of dimensions for the output attribute space.




(a) (b)

Figure 3: (a) An example of a 2-dimensional output dataset partitioned into $3 \times 3$ output tiles, and input chunk i1, i2 and i3, whose mid-points fall inside output tile $t$. Input chunk i1, i2 and i3 map to one, two and four output tiles, respectively. (b) An alternative way to partition an output tile, based on the mid-points of the input chunks. $x_j$ and $y_j$, for $j \in \{0, 1\}$, are the extents of an output tile and an input chunk along dimension $j$, respectively.

$x_j$ where $j = 0, 1, \ldots, d-1$: the extent of an output tile along dimension $j$ of the output attribute space.

$y_j$ where $j = 0, 1, \ldots, d-1$: the extent of an input chunk along dimension $j$ of the output attribute space.

$z_j$ where $j = 0, 1, \ldots, d-1$: the extent of an output chunk along dimension $j$ of the output attribute space.

$A_s$ : the average number of input chunks that map to an output tile under strategy $s$.

$B_s$ : the average number of output chunks assigned to an output tile under strategy $s$.

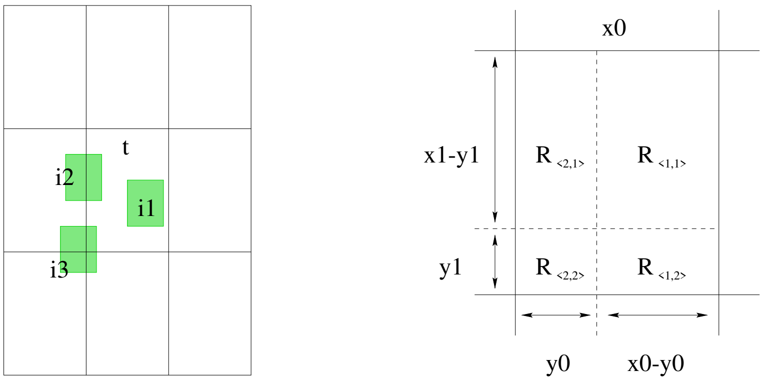
$T_s$ : the total number of output tiles under strategy $s$.

The values of many parameters can be obtained by accessing the index of the input and output datasets. In practice, averages are used when single values cannot be assumed for $i$, $u$, $\alpha$, $\beta$, $y_j$ and $z_j$.

### 3.3 Partitioning MBR of An Output Tile Into Regions

For analysis purpose, the MBR of an output tile is partitioned into several regions. Figure 3(a) shows an example of a 2-dimensional output dataset partitioned into $3 \times 3$ output tiles with three input chunks. The MBRs of the output tiles are shown as white rectangles, while the MBRs of the input chunks are shown as the shaded rectangles. In this example, input chunk i1 maps to one output tile, i2 maps to two output tiles, and i3 maps to four output tiles. Let's consider all the input chunks in $I$ whose *mid-points* fall inside an output tile, such as i1, i2 and i3 of Figure 3(a) falling inside output tile $t$, and group those input chunks by the number of output tiles they map to. Assuming that an input chunk has a smaller extent than that of an output tile, the grouping of the input chunks implies a partitioning of the MBR of output tile $t$ into three regions, $R_1$, $R_2$ and $R_4$, such that all input chunks whose mid-points fall inside region $R_j$ map to exactly $j$ output tiles.

An alternative way to partition the MBR of output tile $t$ is by considering all the input chunks in $I$ whose *top-right corners* fall inside $t$. Figure 4(a) shows an example of a 2-dimensional output dataset partitioned into $3 \times 3$ output tiles, with three input chunks whose MBRs are shown as shaded rectangles. In this example, input chunk i1 maps to one output tile, i2 maps to two output tiles, with two along the first




(a) (b)

Figure 4: (a) An example of a 2-dimensional output dataset partitioned into $3 \times 3$ output tiles, and input chunk i1, i2 and i3, whose top-right corners fall inside output tile $t$. Input chunk i1, i2 and i3 map to one, two and four output tiles, respectively. (b) Partitioning of output tile $t$ into four regions when the extents of input chunks are smaller than that of an output tile.

dimension (ie the horizontal dimension) and one along the second dimension (ie the vertical dimension), and i3 maps to four output tiles, with two along each of the two dimensions. Let's consider all the input chunks in $I$ whose top-right corners fall inside an output tile, such as i1, i2 and i3 of Figure 4(a) falling inside output tile $t$, and group those input chunks by the number of output tiles they map to *along each dimension*. Assuming that an input chunk has a smaller extent than that of an output tile, the grouping of the input chunks implies a partitioning of output tile $t$ into four regions, $\mathcal{R}_{<1,1>}$, $\mathcal{R}_{<1,2>}$, $\mathcal{R}_{<2,1>}$ and $\mathcal{R}_{<2,2>}$, such that all input chunks whose top-right corners fall inside region $\mathcal{R}_{<j,k>}$ map to exactly $j \times k$ output tiles: $j$ output tiles along dimension 0 and $k$ output tiles along dimension 1. These regions are shown in Figure 4(b). For example, in Figure 4(a), input chunk i2 belongs to region $\mathcal{R}_{<2,1>}$, while i3 belongs to region $\mathcal{R}_{<2,2>}$. In general, a $d$-dimensional output tile can be partitioned into $2^d$ regions, each of which labeled as $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$, and an input chunk whose top-right corner falls inside region $\mathcal{R}_v$ maps to $v_j$ output tiles along dimension $j$, for $j = 0, 1, \ldots, d-1$. Since we assume that the input chunks are randomly distributed in the output space, the ratio between the volume of region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$ and the total volume of an output tile can be used as an estimate for the probability that an input chunk would map to $v_j$ output tiles along dimension $j$, for $j = 0, 1, \ldots, d-1$. In the rest of this subsection, we derive the volumes for these regions.

Note that region $\mathcal{R}_{<1,1>}$ in Figure 4(b) corresponds to region $R_1$ in Figure 3, $\mathcal{R}_{<1,2>}$ and $\mathcal{R}_{<2,1>}$ together correspond to $R_2$, and $\mathcal{R}_{<2,2>}$ corresponds to $R_4$. Although the two approaches of partitioning an output tile generate different regions, each pair of corresponding regions from the two approaches have the same volume. Since the approach based on the top-right corners of input chunks extends more naturally to the scenario where input chunks have larger extents than that of an output tile, we will refer to regions generated by this approach during discussion in the remaining of this paper.

Assume that the $d$-dimensional output grid is partitioned regularly into rectangular output tiles and there are $B$ output chunks per output tile. Let each output chunk have a minimum bounding rectangle (MBR) of size $z_j$ along dimension $j = 0, 1, \ldots, d-1$. Then, the extent of the MBR for an output tile in each dimension can be computed as $x_j = z_j n_j$ for $j = 0, 1, \ldots, d-1$, where $n_j$ is the number of output chunks along dimension $j$ of the output tile. In our analysis, we will assume that $n_0 = n_1 = \cdots = n_{d-1}$, and therefore we have $n_j = \sqrt[d]{B}$. We now derive the volumes of the regions described earlier. We first consider the scenario where the input chunks have smaller extent than that of an output tile, and later consider the scenario where the input chunks have larger extent than that of an output tile.

**Small Input Chunks:** (ie $x_j \geq y_j, \forall j = 0, 1, \ldots, d-1$)
$d = 2$**:** (see Figure 4(b))

$$\begin{aligned}
\text{vol}(\mathcal{R}_{<1,1>}) &= (x_0 - y_0)(x_1 - y_1) \\
\text{vol}(\mathcal{R}_{<1,2>}) &= (x_0 - y_0)y_1 \\
\text{vol}(\mathcal{R}_{<2,1>}) &= y_0(x_1 - y_1) \\
\text{vol}(\mathcal{R}_{<2,2>}) &= y_0 y_1
\end{aligned}$$

$d = 3$**:**

$$\begin{aligned}
\text{vol}(\mathcal{R}_{<1,1,1>}) &= (x_0 - y_0)(x_1 - y_1)(x_2 - y_2) \\
\text{vol}(\mathcal{R}_{<1,1,2>}) &= (x_0 - y_0)(x_1 - y_1)y_2 \\
\text{vol}(\mathcal{R}_{<1,2,1>}) &= (x_0 - y_0)y_1(x_2 - y_2) \\
\text{vol}(\mathcal{R}_{<1,2,2>}) &= (x_0 - y_0)y_1 y_2 \\
\text{vol}(\mathcal{R}_{<2,1,1>}) &= y_0(x_1 - y_1)(x_2 - y_2) \\
\text{vol}(\mathcal{R}_{<2,1,2>}) &= y_0(x_1 - y_1)y_2 \\
\text{vol}(\mathcal{R}_{<2,2,1>}) &= y_0 y_1(x_2 - y_2) \\
\text{vol}(\mathcal{R}_{<2,2,2>}) &= y_0 y_1 y_2
\end{aligned}$$

Let function $\Gamma(s, t, j, S)$ be defined as follows.

$$\Gamma(s, t, j, S) = \begin{cases} s & \text{if } j \in S \\ t & \text{otherwise} \end{cases}$$

where $s$ and $t$ are scalars, $j$ is an integer and $S$ is a set of integers. That is, for a given set of integers $S$, $\Gamma(s, t, j, S)$ returns $s$ if $j$ belongs to set $S$; otherwise, it returns $t$. In general, region $\mathcal{R}_{<v_0, v_1, \ldots, v_{d-1}>}$ is a $d$-dimensional hyper-box. Since the input chunks are assumed to have smaller extents than that of an output tile, an input chunk can only map to one or two output tiles along any particular dimension. Hence, we have $v_j \in \{1, 2\}$. As suggested by Figure 4(a) when $d = 2$, the extent of region $\mathcal{R}_{<v_0, v_1, \ldots, v_{d-1}>}$ along dimension $j$ is either $x_j - y_j$ when $v_j = 1$, or $y_j$ when $v_j = 2$. Therefore, the volume of region $\mathcal{R}_{<v_0, v_1, \ldots, v_{d-1}>}$ can be computed as follows.

$$\text{vol}(\mathcal{R}_{<v_0, v_1, \ldots, v_{d-1}>}) = \prod_{j=0}^{d-1} \Gamma(x_j - y_j, y_j, v_j, \{1\}) \tag{1}$$

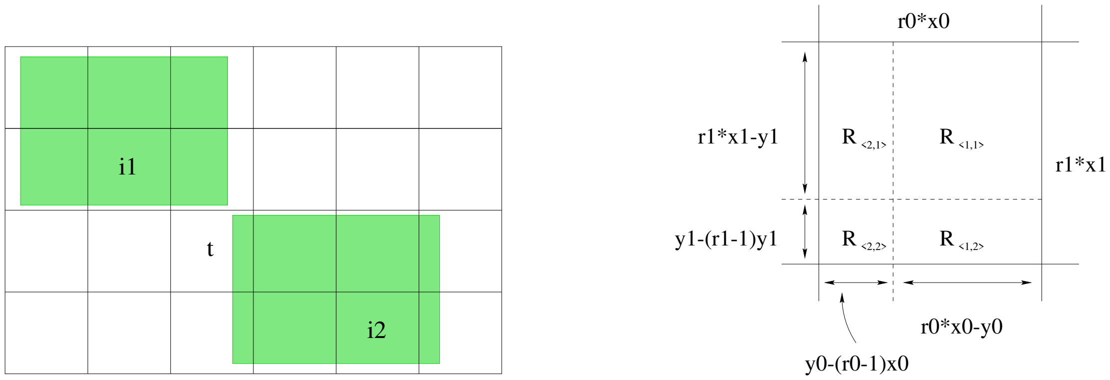
Note that there are $2^d$ regions in total.

**Large Input Chunks:** (ie $\exists j \in [0, d-1]$, such that $x_j \leq y_j$)
Let $r_j$ be the smallest number of output tiles along dimension $j$ in the output attribute space that can entirely contain the MBR of an input chunk along that dimension. That is,

$$r_j = \lceil \frac{y_j}{x_j} \rceil \quad \text{for } j = 0, 1, 2, \ldots, d-1$$

Note that this implies the following, for $j = 0, 1, 2, \ldots, d-1$.

$$(r_j - 1)x_j < y_j \leq r_j x_j$$




(a) (b)

Figure 5: (a) An example of an output dataset partitioned into $6 \times 4$ output tiles, and input chunks i1 and i2, whose extents map to six and eight output tiles, respectively. (b) Partitioning of output tile $t$ into four regions when the extents of input chunks are larger than that of an output tile.

$$r_j \geq 1$$

Figure 5(a) shows the output dataset partitioned into $6 \times 4$ output tiles, and two input chunks with extents larger than that of an output tile mapping to six and eight output tiles, respectively. In this example, we have $r_0 = 3$ and $r_1 = 2$. Figure 5(b) shows how output tile $t$ is partitioned into four regions according to the same way of grouping input tiles whose top-right corners fall inside tile $t$ that was used earlier.

We now compute the volume for region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$, Let's first consider the case where the output space is 2-dimensional.

$$\begin{aligned}
\mathrm{vol}(\mathcal{R}_{<r_0,r_1>}) &= (r_0x_0 - y_0)(r_1x_1 - y_1) \\
\mathrm{vol}(\mathcal{R}_{<r_0+1,r_1>}) &= [y_0 - (r_0 - 1)x_0](r_1x_1 - y_1) \\
\mathrm{vol}(\mathcal{R}_{<r_0,r_1+1>}) &= (r_0x_0 - y_0)[y_1 - (r_1 - 1)x_1] \\
\mathrm{vol}(\mathcal{R}_{<r_0+1,r_1+1>}) &= [y_0 - (r_0 - 1)x_0][y_1 - (r_1 - 1)x_1]
\end{aligned}$$

Extending the analysis carried out for a 2-dimensional attribute space to a $d$-dimensional output space, we have the following.

$$\begin{aligned}
\mathrm{vol}(\mathcal{R}_{<r_0,r_1,\ldots,r_{d-1}>}) &= (r_0x_0 - y_0)(r_1x_1 - y_1)\cdots(r_{d-1}x_{d-1} - y_{d-1}) \\
\mathrm{vol}(\mathcal{R}_{<r_0+1,r_1,\ldots,r_{d-1}>}) &= [y_0 - (r_0 - 1)x_0](r_1x_1 - y_1)\cdots(r_{d-1}x_{d-1} - y_{d-1}) \\
\mathrm{vol}(\mathcal{R}_{<r_0,r_1,\ldots,r_{k-1},r_k+1,r_{k+1},\ldots,r_{d-1}>}) &= (r_0x_0 - y_0)(r_1x_1 - y_1)\cdots(r_{k-1}x_{k-1} - y_{k-1}) \\
&\quad [y_k - (r_k - 1)x_k](r_{k+1}x_{k+1} - y_{k+1})\cdots \\
&\quad (r_{d-1}x_{d-1} - y_{d-1}) \\
&\vdots \\
\mathrm{vol}(\mathcal{R}_{<r_0,r_1,\ldots,r_{k-1},r_k+1,r_{k+1},\ldots,r_{j-1},r_j+1,r_{j-1},\cdots,r_{d-1}>}) &= (r_0x_0 - y_0)(r_1x_1 - y_1)\cdots(r_{k-1}x_{k-1} - y_{k-1}) \\
&\quad [y_k - (r_k - 1)x_k](r_{k+1}x_{k+1} - y_{k+1})\cdots \\
&\quad (r_{j-1}x_{j-1} - y_{j-1})[y_j - (r_j - 1)x_j] \\
&\quad (r_{j+1}x_{j+1} - y_{j+1})\cdots(r_{d-1}x_{d-1} - y_{d-1}) \\
&\vdots
\end{aligned}$$

$$\begin{aligned}\mathrm{vol}(\mathcal{R}_{<r_0+1,r_1+1,\ldots,r_{d-1}+1>}) &= [y_0-(r_0-1)x_0][y_1-(r_1-1)x_1]\cdots\\ &\quad [y_{d-1}-(r_{d-1}-1)x_{d-1}]\end{aligned}$$

Note that an input chunk can only map to $r_j$ or $r_j+1$ output tiles along a particular dimension. Therefore, we have $v_j \in \{r_j, r_j+1\}$, and there are $2^d$ regions in total. Let $\Psi$ be the set of regions from partitioning output tile $t$.

In general, the extent of region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$ along dimension $j$ is either $r_jx_j - y_j$ when $v_j = r_j$, or $y_j - (r_j-1)x_j$ when $v_j = r_j+1$. The volume of region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$, can therefore be computed as follows.

$$\mathrm{vol}(\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}}>) = \prod_{j=0}^{d-1}\Gamma(r_jx_j-y_j, y_j-(r_j-1)x_j, v_j, \{r_j\}) \tag{2}$$

Note that with $r_0 = r_1 = \cdots = r_{d-1} = 1$, Equation (2) becomes Equation (1). Therefore, in the rest of this paper, we will use Equation (2) to compute the volume of a region for both scenarios.

As discussed earlier, the ratio between the volume of region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$ and the total volume of an output tile can be used as an estimate for the probability that an input chunk would map to $v_j$ output tiles along dimension $j$. By the definition of $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$, an input chunk that belongs to region $\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>}$ maps to $\prod_{j=0}^{d-1} v_j$ output tiles, Therefore, the expected number of output tiles that an input chunk maps to, $\lambda$, can be computed as follows.

$$\begin{aligned}\lambda &= \sum_{\mathcal{R}_{<v_0,v_1,\cdots,v_{d-1}>}\in\Psi}\left[\frac{\mathrm{vol}(\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>})}{\mathrm{vol(an\ output\ tile)}}(v_0v_1\cdots v_{d-1})\right]\\ &= \frac{1}{\mathrm{vol(an\ output\ tile)}}\sum_{\mathcal{R}_v\in\Psi}\left[\mathrm{vol}(\mathcal{R}_{<v_0,v_1,\ldots,v_{d-1}>})\,v_0v_1\cdots v_{d-1}\right]\\ &= \frac{1}{x_0x_1\cdots x_{d-1}}\sum_{\mathcal{R}_v\in\Psi}\left\{v_0v_1\cdots v_{d-1}\prod_{j=0}^{d-1}\Gamma(r_jx_j-y_j, y_j-(r_j-1)x_j, v_j, \{r_j\})\right\}\end{aligned} \tag{3}$$

# 4 Analytical Cost Models

In this section we present analytical models to predict the average operation counts of the three query processing strategies. In particular, our models estimate the number of I/O, communication, and computation operations that must be performed by an processor for an output tile in each of the query processing phases (initialization, local reduction, global combine and output handling).

Table 1 shows the expected average number of operations per processor for a tile in each phase. In the remaining of this section, we describe the methods used to compute the expected number of operations. The main assumption of the analytical models described in this paper is that the distribution of the input chunks in the output attribute space must be uniform, and the output dataset must be a regular $d$-dimensional dense array.

## 4.1 Computing Operation Counts for FRA

The number of output tiles and the average number of output chunks in an output tile depend on the aggregate system memory that can be effectively utilized by a query processing strategy. Since an output chunk is replicated across all processors for FRA, the effective system memory for FRA is the size of memory on a

| Query Execution Phase | Query Processing Strategy | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | FRA | | | SRA | | | DA | | |
| | I/O | Comm. | Comp. | I/O | Comm. | Comp. | I/O | Comm. | Comp. |
| Initialization | $\frac{B_{fra}}{P}$ | $\frac{B_{fra}}{P}(P-1)$ | $B_{fra}$ | $\frac{B_{sra}}{P}$ | $g$ | $\frac{B_{sra}}{P}+g$ | $\frac{B_{da}}{P}$ | 0 | $\frac{B_{da}}{P}$ |
| Local Reduction | $\frac{A_{fra}}{P}$ | 0 | $\frac{B_{fra}}{P}\beta$ | $\frac{A_{sra}}{P}$ | 0 | $\frac{B_{sra}}{P}\beta$ | $\frac{A_{da}}{P}$ | $I_{msg}$ | $\frac{B_{da}}{P}\beta$ |
| Global Combine | 0 | $\frac{B_{fra}}{P}(P-1)$ | $\frac{B_{fra}}{P}(P-1)$ | 0 | $g$ | $g$ | 0 | 0 | 0 |
| Output Handling | $\frac{B_{fra}}{P}$ | 0 | $\frac{B_{fra}}{P}$ | $\frac{B_{sra}}{P}$ | 0 | $\frac{B_{sra}}{P}$ | $\frac{B_{da}}{P}$ | 0 | $\frac{B_{da}}{P}$ |

Table 1: The expected average number of I/O, communication, and computation operations per processor for a tile in each phase. $B_{fra}$, $B_{sra}$, and $B_{da}$ denote the expected average number of output chunks per tile for the FRA, SRA, and DA strategies, respectively. Similarly, $A_{fra}$, $A_{sra}$, and $A_{da}$ are the expected average number of input chunks retrieved per tile for the FRA, SRA, and DA strategies. $g$ is the expected average number of ghost chunks per processor for a tile in SRA, and $I_{msg}$ is the expected average number of messages per processor for input chunks in a tile for DA. The average number of output chunks that an input chunk maps to is denoted by $\alpha$, and $\beta$ represents the average number of input chunks that map to an output chunk. $P$ is the number of processors executing the query.

single processor, $M$. Hence, the average number of output chunks per output tile, $B_{fra}$, and the number of output tiles, $T_{fra}$, can be computed as follows.

$$B_{fra} = \frac{M}{u}$$
$$T_{fra} = \frac{U}{B_{fra}} = \frac{Uu}{M}$$

The expected extent of an output tile along dimension $j$ is computed as $x_j = \sqrt[d]{B_{fra}}\, z_j$, for $j = 0, 1, \ldots, d-1$. Equation (3) (see Section 3.3) computes the expected number of output tiles that an input chunk intersects. Therefore, the expected number of input chunks out of $I$ input chunks that map to a given output tile can be computed as follows.

$$A_{fra} = \frac{\lambda I}{T_{fra}}$$

Assuming perfect declustering, each processor reads $\frac{B_{fra}}{P}$ output chunks during the initialization phase, and $\frac{A_{fra}}{P}$ input chunks during the local reduction phase. Each output chunk is sent to $P-1$ processors, therefore each processor sends out $\frac{B_{fra}}{P}(P-1)$ output chunks during the initialization phase and $\frac{B_{fra}}{P}(P-1)$ output chunks during the global combine phase. Since each output chunk is mapped to by $\beta$ input chunks, $B_{fra}\beta$ computation operations are carried out in total for an output tile of $B_{fra}$ output chunks. Assuming perfect declustering of the input chunks across all processors, each processor is responsible for $\frac{B_{fra}\beta}{P}$ computation operations per output tile.

### 4.2 Computing Operation Counts for SRA

Let $e$ be the average percent of system memory used for local output chunks in an output tile. That is, if $g$ is the average number of ghost chunks per processor per output tile, we have

$$e = \frac{b_{sra}}{b_{sra} + g}$$

where $b_{sra}$ is the average number of local output chunks per processor per output tile. Note that we have $\frac{1}{p} \leq e \leq 1$. When $e$ is equal to $\frac{1}{p}$, SRA is equivalent to FRA and $g = b_{sra}(P-1)$. Given the value of $e$, we have the following.

$$\begin{aligned} B_{sra} &= \frac{ePM}{u} \\ b_{sra} &= \frac{B_{sra}}{P} = \frac{eM}{u} \\ T_{sra} &= \frac{U}{B_{sra}} = \frac{Uu}{ePM} \end{aligned}$$

We compute $g$ and $e$ as follows. The goal of the declustering algorithms used in ADR [5, 9] is to achieve good I/O parallelism when retrieving input and output chunks from disks. To achieve this goal, the algorithms distribute spatially close chunks evenly across as many processors as possible. Therefore, $\beta$ input chunks that map to an output chunk on processor $p$ can be expected to be distributed across as many processors as possible. Let $g'$ be the average number of ghost chunks that are created for an output chunk. Then, with $b_{sra}$ local output chunks per processor in an output tile, on average a processor creates a total of $g = b_{sra}g'$ ghost chunks per output tile, and $P$ processors create $Pb_{sra}g'$ ghost chunks per output tile.

Under the assumption that input chunks that map to the same output chunk are distributed across as many processors as possible, SRA becomes FRA if $\beta \geq P$. When $\beta < P$, we have

$$\begin{aligned} g' &= \text{prob}\{p \text{ is one of } \beta \text{ procs}\}(\beta - 1) + \text{prob}\{p \text{ is not one of } \beta \text{ procs}\}\beta \\ &= \frac{\beta}{P}(\beta - 1) + (1 - \frac{\beta}{P})\beta \\ &= \frac{P-1}{P}\beta \end{aligned}$$

And hence,

$$\begin{aligned} e &= \frac{b_{sra}}{b_{sra} + g} = \frac{b_{sra}}{b_{sra} + b_{sra}g'} = \frac{1}{1 + g'} = \frac{P}{P + \beta(P-1)} \\ g &= b_{sra}g' = b_{sra}\frac{P-1}{P}\beta \end{aligned}$$

Similar to FRA, the expected extent of an output tile along the dimension $j$ is computed as $x_j = \sqrt[d]{B_{sra}}\, z_j$, for $j = 0, 1, \ldots, d-1$, and $\lambda$ is computed by Equation (3). The expected number of input chunks out of $I$ input chunks that intersect with a given output tile therefore can be computed as follows.

$$A_{sra} = \frac{\lambda I}{T_{sra}}$$

Assuming perfect declustering, each processor reads $b_{sra} = \frac{B_{sra}}{P}$ output chunks during the initialization phase, and $\frac{A_{sra}}{P}$ input chunks during the local reduction phase. Each processor sends $g$ messages for output chunks during the initialization phase, and $g$ messages for output chunks during the global combine phase.

Similar to FRA, since each output chunk is mapped to by $\beta$ input chunks, $B_{sra}\beta$ computation operations are carried out in total for an output tile of $B_{sra}$ output chunks. Assuming perfect declustering of the input chunks across all processors, each processor is responsible for $\frac{B_{sra}\beta}{P}$ computation operations per output tile.

### 4.3 Computing Operation Counts for DA

For DA, the output chunks are not replicated, so the effective overall system memory is $P \times M$. Therefore, the average number of output chunks and the number of output tiles can be computed as follows.

$$\begin{aligned} B_{da} &= \frac{PM}{u} \\ T_{da} &= \frac{U}{B_{da}} = \frac{Uu}{PM} \end{aligned}$$

Similar to FRA, the expected extent of an output tile along dimension $j$, $x_j$ for $j = 0, 1, \ldots, d-1$, and the expected number of input chunks out of $I$ input chunks that map to a given output tile, $A_{da}$, can be computed as follows.

$$\begin{aligned} x_j &= \sqrt[d]{B_{da}}\, z_j \quad \text{for } j = 0, 1, \ldots, d-1 \\ A_{da} &= \frac{\lambda I}{T_{da}} \end{aligned}$$

During the local reduction phase for DA, local input chunks that map to output chunks on other processors must be sent to those processors. As a result, DA requires interprocessor communication for input chunks. We estimate the number of messages for input chunks for each processor, $I_{msg}$, as follows.

1. Compute for each region $\mathcal{R}_v \in \Psi$ of an output tile (see Section 3.3) the expected number of messages generated for an input chunk that belongs to $\mathcal{R}_v$. This is denoted as $I_v$.

2. Compute $I_{msg}$ as the sum of all $I_v$'s, weighted by the volumes of the regions that correspond to the $I_v$'s.

$$I_{msg} = \sum_{\mathcal{R}_v \in \Psi} \frac{\text{vol}(\mathcal{R}_v)}{\text{vol(an output tile)}} I_v$$

Equation (2) in Section 3.3 computes the volume of region $\mathcal{R}_v$. We now compute $I_v$, the expected number of messages for an input chunk in region $\mathcal{R}_v \in \Psi$. First, let $\mathcal{C}$ be defined as follows.

$$\begin{aligned} \mathcal{C}(k, P) &= \begin{cases} P - 1 & \text{if } k \geq P \\ (k-1)\frac{k}{P} + k(1 - \frac{k}{P}) & \text{otherwise} \end{cases} \\ &= \begin{cases} P - 1 & \text{if } k \geq P \\ \frac{P-1}{P} k & \text{otherwise} \end{cases} \end{aligned}$$

For an input chunk that maps to $k$ output chunks in an output tile, under perfect declustering, those $k$ output chunks are expected to be distributed to as many processors as possible. That is, if $k \geq P$, then the $k$ mapped output chunks are stored on $P$ processors; otherwise, they are stored on $k$ processors. $\mathcal{C}(k, P)$ therefore gives the expected number of *remote* processors that own at least one of the $k$ mapped output chunks. Note that this is also the expected number of messages that are generated in DA for an input chunk that maps to $k$ output chunks in an output tile.

We now continue the computation of $I_v$ for region $\mathcal{R}_v$. First, consider the scenario where input chunks have extents smaller than that of an output tile. See Figure 4(b) for the four regions of an output tile in this scenario. To simplify the analysis, for each of the four regions, we designate an input chunk as a *representative input chunk* for all input chunks in the same region, and use the expected number of messages that are generated by DA for the representative input chunk as the expected number of messages generated for each input chunk in that region.

First consider region $\mathcal{R}_{<1,1>}$ in Figure 4(b). Since all input chunks in region $\mathcal{R}_{<1,1>}$ map to only one output tile, they are all expected to generate the same number of messages. With each input chunk mapping to $\alpha$ output chunks, we have

$$I_{<1,1>} = \mathcal{C}(\alpha, P)$$

Now let's consider an input chunk $c$ in region $\mathcal{R}_{<1,2>}$, where $c$ maps to two output tiles, say $t$ and $s$. Under the assumption that the output dataset is a regular 2-dimensional dense array, the number of output chunks that input chunk $c$ maps to in output tile $t$ is proportional to the volume (or area when $d = 2$) of the portion of $c$ that falls inside output tile $t$. Suppose that the volume of the portion of input chunk $c$ contained in output tile $t$ is $\omega_1$, and the volume of the portion of $c$ contained in output tile $s$ is $\omega_2$, and vol($c$) = $\omega_1 + \omega_2$. Then the expected number of messages generated for input chunk $c$ can be computed as the sum of the expected number of messages for $c$ in output tile $t$ and the expected number of messages for $c$ in output tile $s$, which is equal to $\mathcal{C}(\frac{\omega_1}{\omega_1+\omega_2}\alpha, P) + \mathcal{C}(\frac{\omega_2}{\omega_1+\omega_2}\alpha, P)$.

Imagine the top-right corner of input chunk $c$ slides from the top towards the bottom of region $\mathcal{R}_{<1,2>}$ in output tile $t$. Initially, most of the $\alpha$ output chunks that $c$ maps to belong to $t$. As $c$ slides towards the bottom of region $\mathcal{R}_{<1,2>}$ in $t$, $c$ overlaps with $s$, and therefore some of the $\alpha$ output chunks now belong to $t$. When its top-right corner is located at the mid-point of region $\mathcal{R}_{<1,2>}$, input chunk $c$ is evenly split between $t$ and $s$. As a result, half of the $\alpha$ output chunks that $c$ maps to belong to output tile $t$, and half of them belong to output tile $s$. As its top-right corner moves passed the mid-point of region $\mathcal{R}_{<1,2>}$ towards the bottom of the region in $t$, more and more of the $\alpha$ output chunks that $c$ maps to belong to output tile $s$. One can obviously see that as $c$ slides from the top towards the bottom of region $\mathcal{R}_{<1,2>}$ in $t$, the expected number of messages for $c$ first increases, maxs out at the mid-point of $\mathcal{R}_{<1,2>}$, and then decreases. To pick a reasonable representative input chunk for region $\mathcal{R}_{<1,2>}$, we choose the input chunk whose top-right corner is located at one-quarter below the top of region $\mathcal{R}_{<1,2>}$ and one-quarter to the left of the right-boundary of region $\mathcal{R}_{<1,2>}$, though the input chunk whose top-right corner is located at one-quarter above the bottom of region $\mathcal{R}_{<1,2>}$ is equally desired. Figure 6 shows the top-right corners of the four representative input chunks for the four regions of an output tile. As discussed earlier, all input chunks in region $\mathcal{R}_{<1,1>}$ generate the same number of messages, and hence any input chunk in that region can be chosen as the representative input chunk. For convenience, we choose the one with its top-right corner located at one-quarter from the top and one-quarter from the right boundaries of region $\mathcal{R}_{<1,1>}$. The top-right corners of the representative input chunks for the other two regions, $\mathcal{R}_{<2,1>}$ and $\mathcal{R}_{<2,2>}$, are chosen in a similar way as for region $\mathcal{R}_{<1,2>}$. The number of messages that these four representative input chunks generate are used for the expected numbers of messages for input chunks in the four regions, and they are computed as follows.

$$\begin{aligned}
I_{<1,1>} &= \mathcal{C}(\alpha, P) \\
I_{<1,2>} &= \mathcal{C}(\frac{y_0 \frac{3}{4} y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{y_0 \frac{1}{4} y_1}{y_0 y_1}\alpha, P) = \mathcal{C}(\frac{3}{4}\alpha, P) + \mathcal{C}(\frac{1}{4}\alpha, P) \\
I_{<2,1>} &= \mathcal{C}(\frac{\frac{3}{4} y_0 y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{1}{4} y_0 y_1}{y_0 y_1}\alpha, P) = \mathcal{C}(\frac{3}{4}\alpha, P) + \mathcal{C}(\frac{1}{4}\alpha, P) \\
I_{<2,2>} &= \mathcal{C}(\frac{\frac{1}{4} y_0 \frac{1}{4} y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{3}{4} y_0 \frac{1}{4} y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{1}{4} y_0 \frac{3}{4} y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{3}{4} y_0 \frac{3}{4} y_1}{y_0 y_1}\alpha, P)
\end{aligned}$$

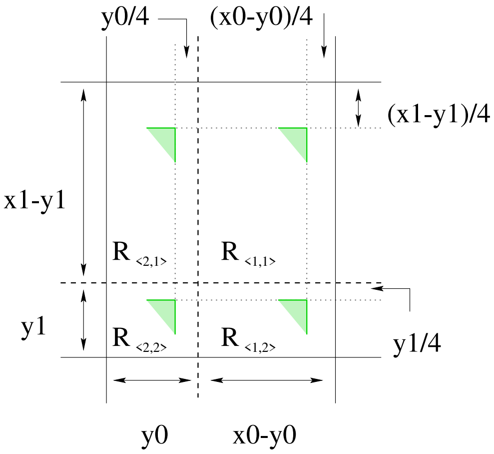



Figure 6: The top-right corners, shown as shaded triangles, of the four representative input chunks for the four regions of an output tile, under the assumption that input chunk extents are smaller than that of an output tile.

$$= \quad \mathcal{C}(\frac{1}{16}\alpha, P) + 2\,\mathcal{C}(\frac{3}{16}\alpha, P) + \mathcal{C}(\frac{9}{16}\alpha, P)$$

Note that the representative chunks in region $\mathcal{R}_{<1,1>}$, $\mathcal{R}_{<1,2>}$, $\mathcal{R}_{<2,1>}$ and $\mathcal{R}_{<2,2>}$ map to one, two, two and four output tiles, respectively. That is why $I_{<1,1>}$, $I_{<1,2>}$, $I_{<2,1>}$ and $I_{<2,2>}$ are computed as sums of one, two, two and four terms, respectively.

Now consider the scenario where input chunks have extents larger than that of an output tile. Let's start with the 2-dimensional case. Similar to the analysis in the scenario where input chunks have small extents, we choose a representative input chunk for each of the four regions shown in Figure 5(b). The top-right corners of those representative input chunks are chosen the same way they were chosen for the smaller input chunk scenario, and Figure 7 shows the four representative input chunks for an example where $r_0 = 3$ and $r_1 = 1$. The expected number of messages for the four representative input chunks for the example in Figure 7 are given below.

$$\begin{aligned}
I_{<r_0,r_1>} &= (r_0-2)\mathcal{C}(\frac{x_0 y_1}{y_0 y_1}, P) + \mathcal{C}(\frac{\frac{y_0-(r_0-4)x_0}{4} y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{3y_0-(3r_0-4)x_0}{4} y_1}{y_0 y_1}\alpha, P) \\
I_{<r_0+1,r_1>} &= (r_0-1)\mathcal{C}(\frac{x_0 y_1}{y_0 y_1}, P) + \mathcal{C}(\frac{\frac{3}{4}[y_0-(r_0-1)x_0]y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{1}{4}[y_0-(r_0-1)x_0]y_1}{y_0 y_1}\alpha, P) \\
I_{<r_0,r_1+1>} &= (r_0-2)\mathcal{C}(\frac{x_0\frac{3}{4}y_1}{y_0 y_1}, P) + (r_0-2)\mathcal{C}(\frac{x_0\frac{1}{4}y_1}{y_0 y_1}, P) \\
&\quad + \mathcal{C}(\frac{\frac{y_0-(r_0-4)x_0}{4}\frac{3}{4}y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{y_0-(r_0-4)x_0}{4}\frac{1}{4}y_1}{y_0 y_1}\alpha, P) \\
&\quad + \mathcal{C}(\frac{\frac{3y_0-(3r_0-4)x_0}{4}\frac{3}{4}y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{3y_0-(3r_0-4)x_0}{4}\frac{1}{4}y_1}{y_0 y_1}\alpha, P) \\
I_{<r_0+1,r_1+1>} &= (r_0-1)\mathcal{C}(\frac{x_0\frac{3}{4}y_1}{y_0 y_1}, P) + (r_0-1)\mathcal{C}(\frac{x_0\frac{1}{4}y_1}{y_0 y_1}, P) \\
&\quad + \mathcal{C}(\frac{\frac{3}{4}[y_0-(r_0-1)x_0]\frac{3}{4}y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{3}{4}[y_0-(r_0-1)x_0]\frac{1}{4}y_1}{y_0 y_1}\alpha, P)
\end{aligned}$$

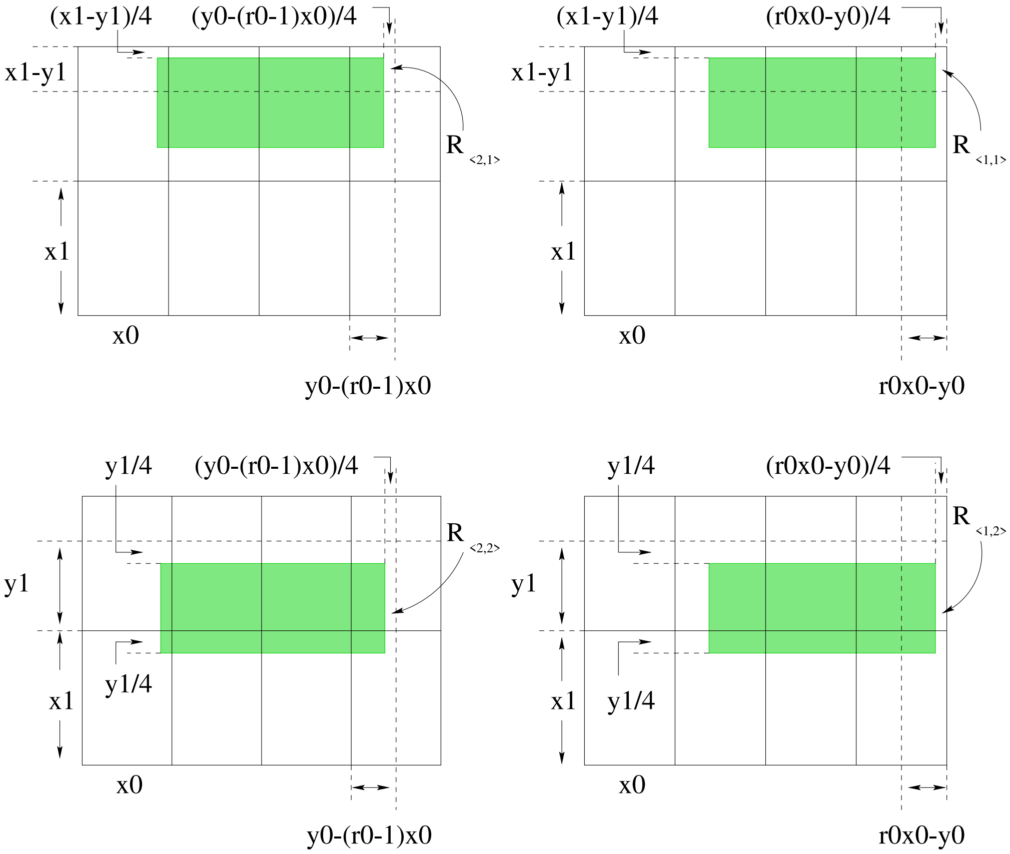



Figure 7: The representative input chunks, shown as the shaded rectangles, for region $\mathcal{R}_{<2,1>}$ (top-left), $\mathcal{R}_{<1,1>}$ (top-right), $\mathcal{R}_{<2,2>}$ (bottom-left) and $\mathcal{R}_{<1,2>}$ (bottom-right), with $r_0 = 3$ and $r_1 = 1$.

$$+\mathcal{C}(\frac{\frac{1}{4}[y_0 - (r_0 - 1)x_0]\frac{3}{4}y_1}{y_0 y_1}\alpha, P) + \mathcal{C}(\frac{\frac{1}{4}[y_0 - (r_0 - 1)x_0]\frac{1}{4}y_1}{y_0 y_1}\alpha, P)$$

For the $d$-dimensional case, one could apply the same analysis for the 2-dimensional case to each of the $d$ dimensions separately, and combine the results from the $d$ dimensions into the final result as follows. All representative input chunks of an output tile, when projected onto dimension $j$, becomes two segments: one corresponds to representative input chunks in regions $\mathcal{R}_{<\ldots,r_j,\ldots>}$, and the other corresponds to representative input chunks in regions $\mathcal{R}_{<\ldots,r_j+1,\ldots>}$. We refer to these two segments as $\mathcal{S}_{r_j}$ and $\mathcal{S}_{r_j+1}$. Depending on the value of $r_j$, each of the two segments would intersect one or many output tiles along dimension $j$, and for each intersection, the length of the overlap between a segment and an output tile determines in part how many messages DA generates for a representative input chunk when processing that output tile. As one will see, for each segment, there are at most three different lengths. Define $e_j(k, l)$, where $k \in \{r_j, r_j + 1\}$ and $l \in \{0, 1, 2\}$, as the three possible lengths of the overlap between $\mathcal{S}_k$ and the output tiles $\mathcal{S}_k$ intersects, and let $c_j(k, l)$ be the number of output tiles that $\mathcal{S}_k$ intersects at the length of $e_j(k, l)$. We now look at two different cases, $r_j = 1$ and $r_j \geq 2$, and for each case, compute $c_j(k, l)$ and $e_j(k, l)$.

**Case 1** $r_j = 1$ (ie $y_j \leq x_j$). Figure 8 shows the two segments obtained by projecting MBRs of the representative input chunks onto dimension $j$ when the extents of input chunks are smaller than that of an output tile.

- Segment $\mathcal{S}_{r_j}$ is entirely contained within one output tile, and hence the length of the overlapping

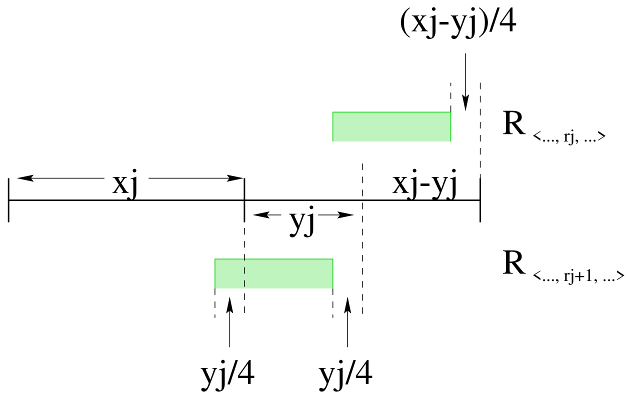



Figure 8: Two segments, $\mathcal{S}_{r_j}$ on top and $\mathcal{S}_{r_j+1}$ at bottom, obtained by projecting the MBRs of all representative input chunks onto dimension $j$, when the input chunk extents are smaller than that of an output tile. The solid line in the middle represents the extents of two output tiles projected onto dimension $j$. Segment $\mathcal{S}_{r_j}$ is entirely contained within one output tile, while segment $\mathcal{S}_{r_j+1}$ intersects two output tiles.

segment between $\mathcal{S}_{r_j}$ and the output tile is $y_j$. Therefore, we have

$$c_j(r_j, l) = \begin{cases} 1 & \text{if } l = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$e_j(r_j, l) = \begin{cases} y_j & \text{if } l = 0 \\ 0 & \text{otherwise} \end{cases}$$

- Segment $\mathcal{S}_{r_j+1}$ intersects one output tile at length $\frac{3}{4}y_j$ and another at length $\frac{1}{4}y_j$. Therefore, we have

$$c_j(r_j + 1, l) = \begin{cases} 1 & \text{if } l = 0, 1 \\ 0 & \text{otherwise} \end{cases}$$

$$e_j(r_j + 1, l) = \begin{cases} \frac{3}{4}y_j & \text{if } l = 0 \\ \frac{1}{4}y_j & \text{if } l = 1 \\ 0 & \text{otherwise} \end{cases}$$

**Case 2:** $r_j \geq 2$ (ie $y_j > x_j$). Figure 9 shows the two segments obtained by projecting the MBRs of the representative input chunks onto dimension $j$ when the extents of input chunks are larger than that of an output tile.

- Segment $\mathcal{S}_{r_j}$ intersects one tile at length $\frac{y_j-(r_j-4)x_j}{4}$, $r_j - 2$ output tiles at length $x_j$, and one output tile at length $\frac{3y_j-(3r_j-4)x_j}{4}$. Therefore, we have

$$c_j(r_j, l) = \begin{cases} 1 & \text{if } l = 0, 2 \\ r_j - 2 & \text{if } l = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$e_j(k, l) = \begin{cases} \frac{y_j-(r_j-4)x_j}{4} & \text{if } l = 0 \\ x_j & \text{if } l = 1 \\ \frac{3y_j-(3r_j-4)x_j}{4} & \text{if } l = 2 \\ 0 & \text{otherwise} \end{cases}$$

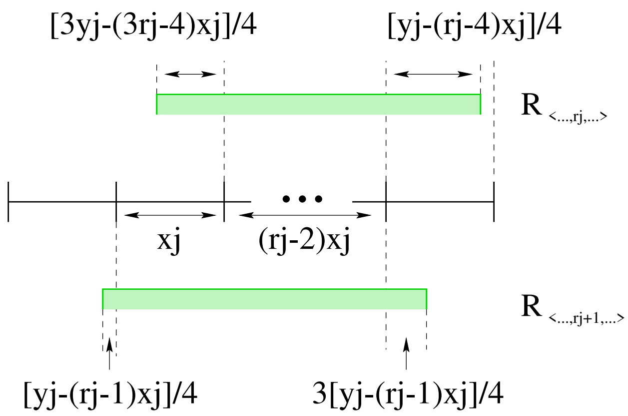



Figure 9: Two segments, $\mathcal{S}_{r_j}$ on top and $\mathcal{S}_{r_j+1}$ at bottom, obtained by projecting the MBRs of all representative input chunks onto dimension $j$, when the input chunk extent is larger than that of an output tile. The solid line in the middle represents the extents of $r_j + 1$ output tiles projected onto dimension $j$. Segment $\mathcal{S}_{r_j}$ intersects $r_j$ output tiles, while segment $\mathcal{S}_{r_j+1}$ intersects $r_j + 1$ output tiles.

| $r_j$ | $c_j, e_j$ | $l = 0$ | $l = 1$ | $l = 2$ |
|---|---|---|---|---|
| $r_j = 1$ | $c_j(r_j, l)$ | 1 | 0 | 0 |
| | $e_j(r_j, l)$ | $y_j$ | 0 | 0 |
| | $c_j(r_j + 1, l)$ | 1 | 1 | 0 |
| | $e_j(r_j + 1, l)$ | $\frac{3}{4}y_j$ | $\frac{1}{4}y_j$ | |
| $r_j \geq 2$ | $c_j(r_j, l)$ | 1 | $r_j - 2$ | 1 |
| | $e_j(r_j, l)$ | $\frac{y_j-(r_j-4)x_j}{4}$ | $x_j$ | $\frac{3y_j-(3r_j-4)x_j}{4}$ |
| | $c_j(r_j + 1, l)$ | 1 | $r_j - 1$ | 1 |
| | $e_j(r_j + 1, l)$ | $\frac{3}{4}[y_j - (r_j - 1)x_j]$ | $x_j$ | $\frac{1}{4}[y_j - (r_j - 1)x_j]$ |

Table 2: Summary of the values for $c_j(k, l)$ and $e_j(k, l)$ where $k = r_j, r_j + 1$ and $l = 0, 1, 2$.

- Segment $\mathcal{S}_{r_j+1}$ intersects one tile at length $\frac{3}{4}[y_0 - (r_j - 1)x_j]$, $r_j - 1$ output tiles at length $x_j$, and one output tile at length $\frac{1}{4}[y_j - (r_j - 1)x_1]$. Therefore, we have

$$c_j(r_j + 1, l) = \begin{cases} 1 & \text{if } l = 0, 2 \\ r_j - 1 & \text{if } l = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$e_j(r_j + 1, l) = \begin{cases} \frac{3}{4}[y_j - (r_j - 1)x_j] & \text{if } l = 0 \\ x_j & \text{if } l = 1 \\ \frac{1}{4}[y_j - (r_j - 1)x_j] & \text{if } l = 2 \\ 0 & \text{otherwise} \end{cases}$$

Table 2 summarizes the values of $c_j(k, l)$ and $e_j(k, l)$. Define $\Theta_d$ as the set of all possible $d$-dimensional vectors where the domain of each element of these vectors is $\{0, 1, 2\}$. That is,

$$\Theta_d = \{< n_0, n_1, \ldots, n_{d-1} > | n_j \in \{0, 1, 2\} \text{ for } j = 0, 1, \ldots, d - 1\}$$

For given $r_0, r_1, \ldots, r_{d-1}$, $I_{<v_0,v_1,\ldots,v_{d-1}>}$ can be computed as follows.

$$I_{<v_0,v_1,\ldots,v_{d-1}>} = \sum_{<\theta_0,\theta_1,\ldots,\theta_{d-1}>\in\Theta_d} \left\{ \left[ \prod_{j=0}^{d-1} c_j(v_j,\theta_j) \right] \mathcal{C}(\alpha[\prod_{j=0}^{d-1} e_j(v_j,\theta_j)], P) \right\}$$

where $v_j \in \{r_j, r_j+1\}$ for $j = 0, 1, 2, \ldots, d-1$. Now that we can compute $I_v$ for each of the $2^d$ regions of an output tile, the expected number of messages for an input chunk, $m$, can be computed as follows.

$$m = \sum_{\mathcal{R}_v\in\Psi} \left\{ \frac{\text{vol}(\mathcal{R}_v)}{x_0 x_1 \cdots x_{d-1}} I_v \right\}$$

where vol($\mathcal{R}_v$) is computed by Equation (2) in Section 3.3. With $A_{da}$ input chunks per output tile, $I_{msg}$, the expected number of input chunk messages for a processor per tile, can be computed as follows.

$$\begin{aligned} I_{msg} &= \frac{A_{da}}{P} m \\ &= \frac{A_{da}}{P} \sum_{\mathcal{R}_v\in\Psi} \left\{ \frac{\text{vol}(\mathcal{R}_v)}{x_0 x_1 \cdots x_{d-1}} I_v \right\} \end{aligned}$$

Assuming perfect declustering, each processor reads $\frac{B_{da}}{P}$ output chunks during the initialization phase, and $\frac{A_{da}}{P}$ input chunks during the local reduction phase. Each processor sends $I_{msg}$ messages for input chunks during the the local reduction phase. Similar to FRA, since each output chunk is mapped to by $\beta$ input chunks, $B_{da}\beta$ computation operations are carried out in total for an output tile of $B_{da}$ output chunks. Assuming perfect declustering of the input chunks across all processors, each processor is responsible for $\frac{B_{da}\beta}{P}$ computation operations per output tile.

# 5 Cost Model Validation

In this section we validate our cost models with queries for synthetic datasets and for several driving applications.

We first use synthetic datasets to evaluate the cost models under controlled scenarios. The output dataset is a 2-dimensional rectangular array. The entire output attribute space is regularly partitioned into non-overlapping rectangles, with each rectangle representing an output chunk in the output dataset. The input dataset has a 3-dimensional attribute space, and input chunks were placed in the input space randomly with a uniform distribution. The assignment of input and output chunks to the disks was done using a Hilbert curve-based declustering algorithm [5]. In these experiments the size of the input and output datasets were fixed. The output dataset size is set at 400MB, with 1600 output chunks. The input dataset size is set at 1.6GBytes. We varied the number and extent of input chunks to produce two sets of queries: one set with $\alpha = \beta$, and the other with $8\alpha = \beta$, where $\alpha$ is the number of output chunks that an input chunk maps to, and $\beta$ is the number of input chunks that an output chunk is mapped to. For each set of queries, $\alpha$ is set to 1.5, 4, 9, and 16. We set the computation time to 1 millisecond for processing an output chunk in the initialization, global combine, and output handling phases, and to 5 milliseconds for processing each intersecting (input,output) chunk pair in the local reduction phase. The number of ADR back-end processors is varied from 8, 16, 32, 64 to 128. For a given number of processors, a query plan is generated for each query under each of the three strategies. Actual operation counts for a given query are obtained by scanning through the query plan and selecting the maximum counts among all the processors, while estimated operation counts per processor are computed from the cost models. These actual and estimated

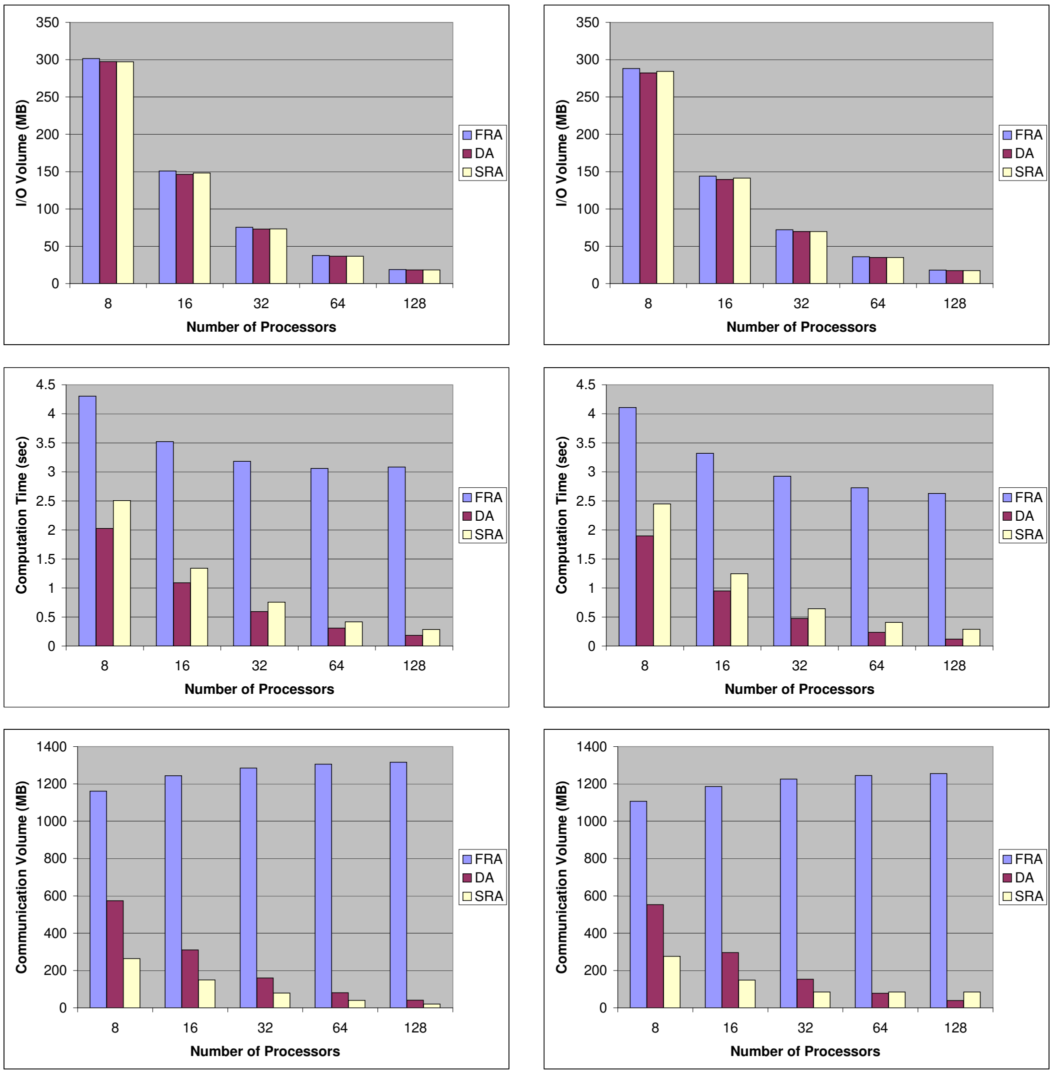



Figure 10: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = \beta = 1.5$.

counts are turned into I/O volume, computation time and communication volume, and compared against each other. Figure 10– 17 show the actual and estimated values for I/O volume, computation time and communication volume for the two sets of synthetic queries. As is seen from the figures, the cost models are able to accurately estimate the I/O volume, computation time and communication volume for the query processing strategies for different $\alpha$ and $\beta$ values for varying number of processors.

Figure 10– 17 show that both FRA and SRA read more data than DA does, and this is because FRA and SRA use part of the system memory for ghost chunks and therefore generates more output tiles than DA does. As a result, input chunks are retrieved from disks more times in FRA and SRA than they are in DA. The figures also show that the computation time decreases for all strategies as the number of processors increases. This is because the computation operations are distributed among all processors and therefore as the number of processors increases, each processor is responsible for less computation. Due to the computation overhead for initializing the ghost chunks during the initialization phase and for combining the ghost chunks during the global combine phase, both FRA and SRA perform more computation than

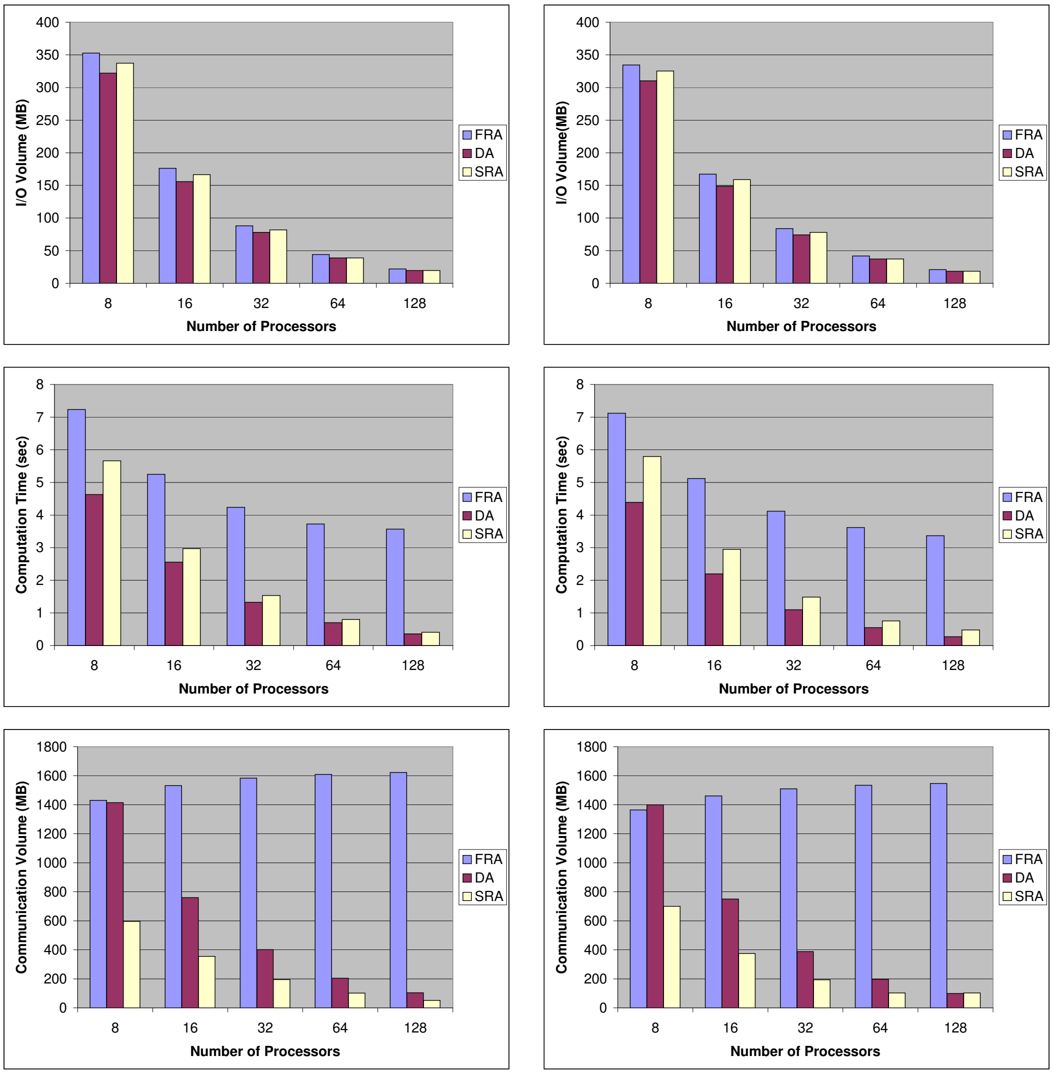



Figure 11: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = \beta = 4$.

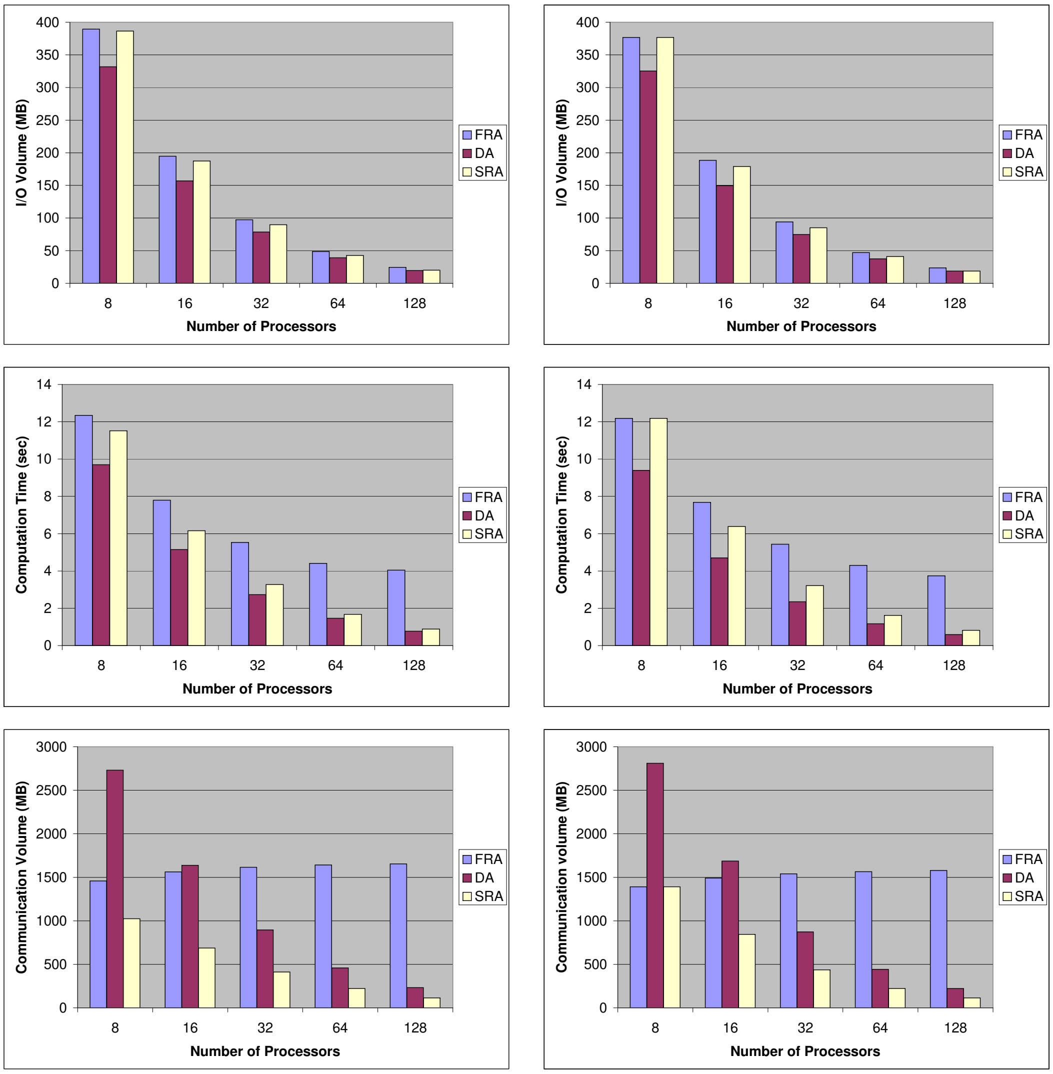



Figure 12: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = \beta = 9$.

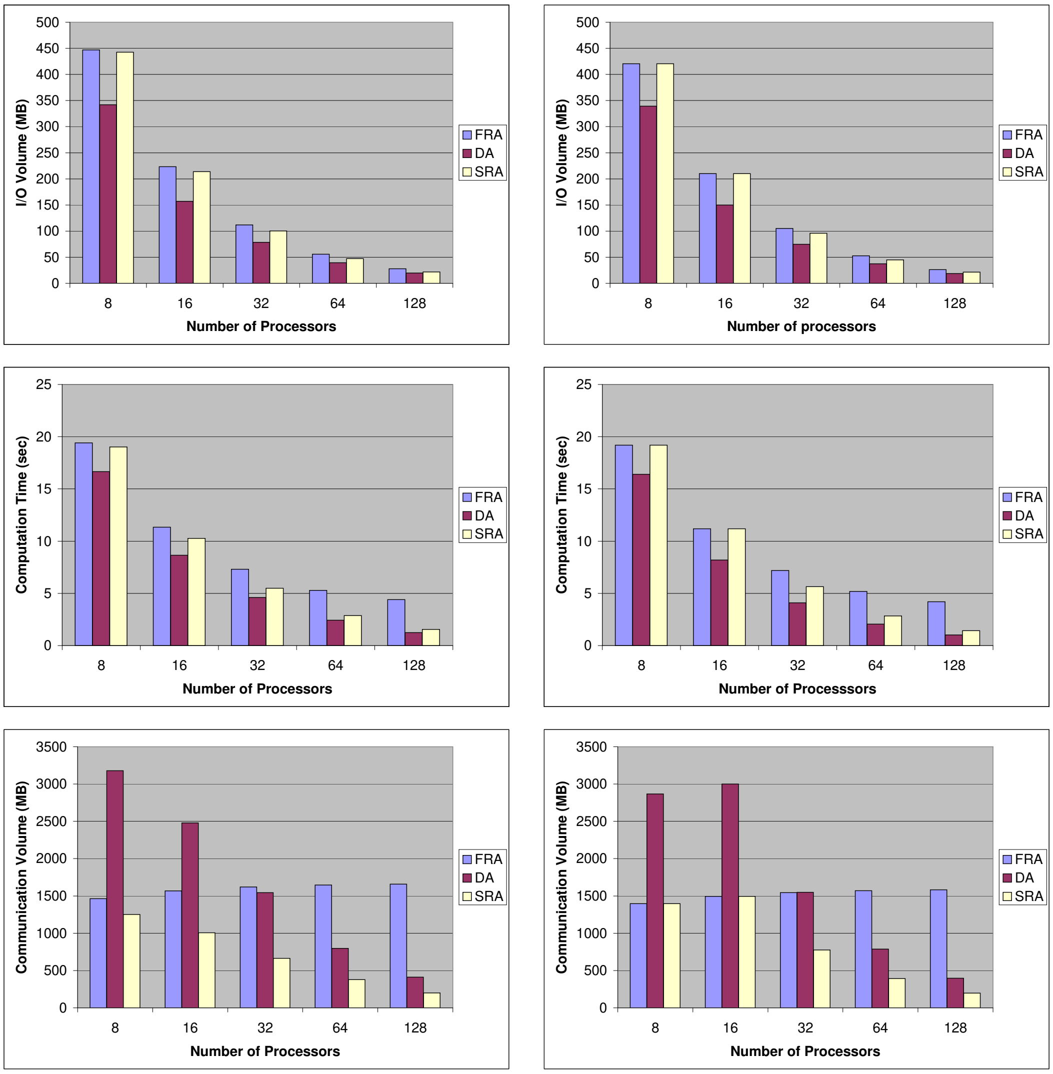



Figure 13: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = \beta = 16$.

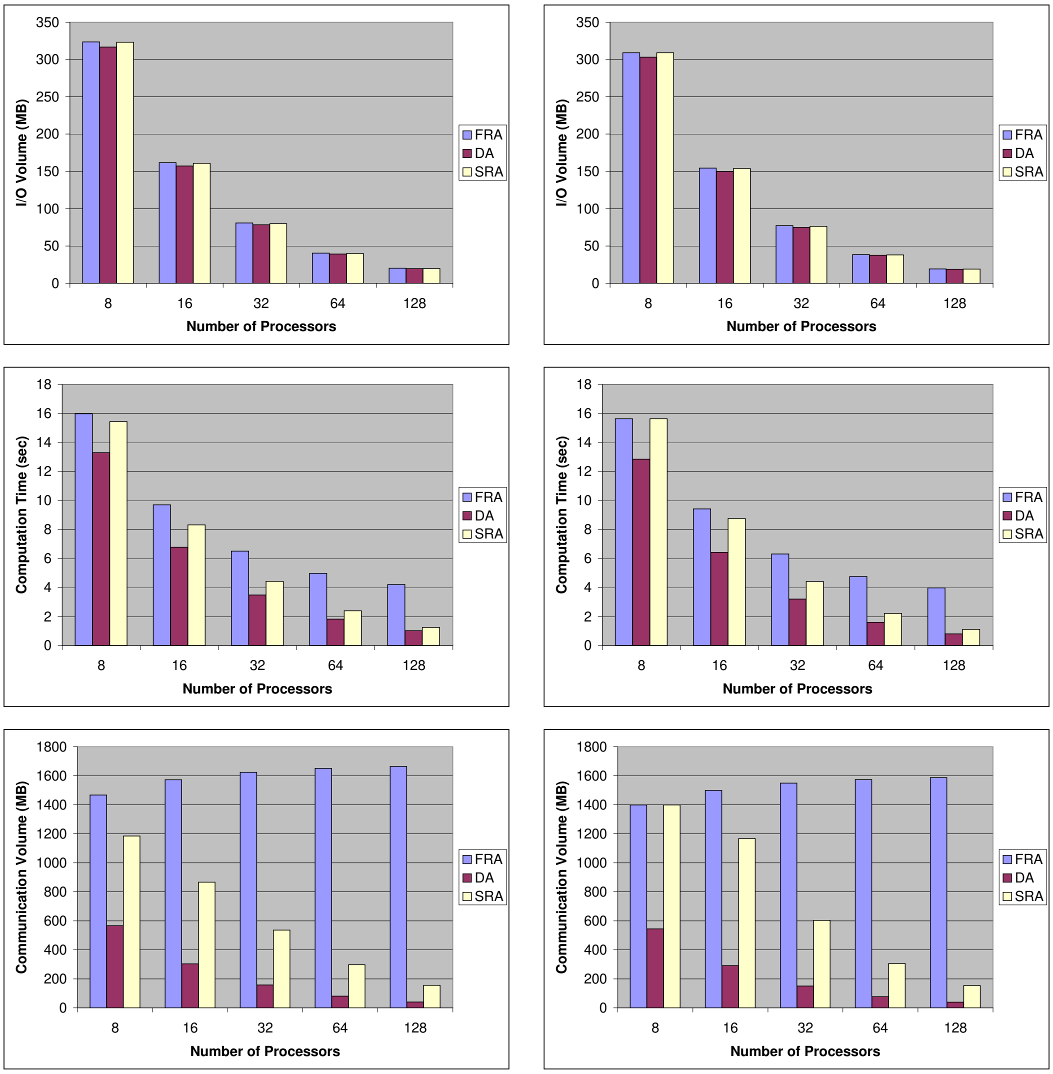



Figure 14: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = 1.5$, $\beta = 12$.

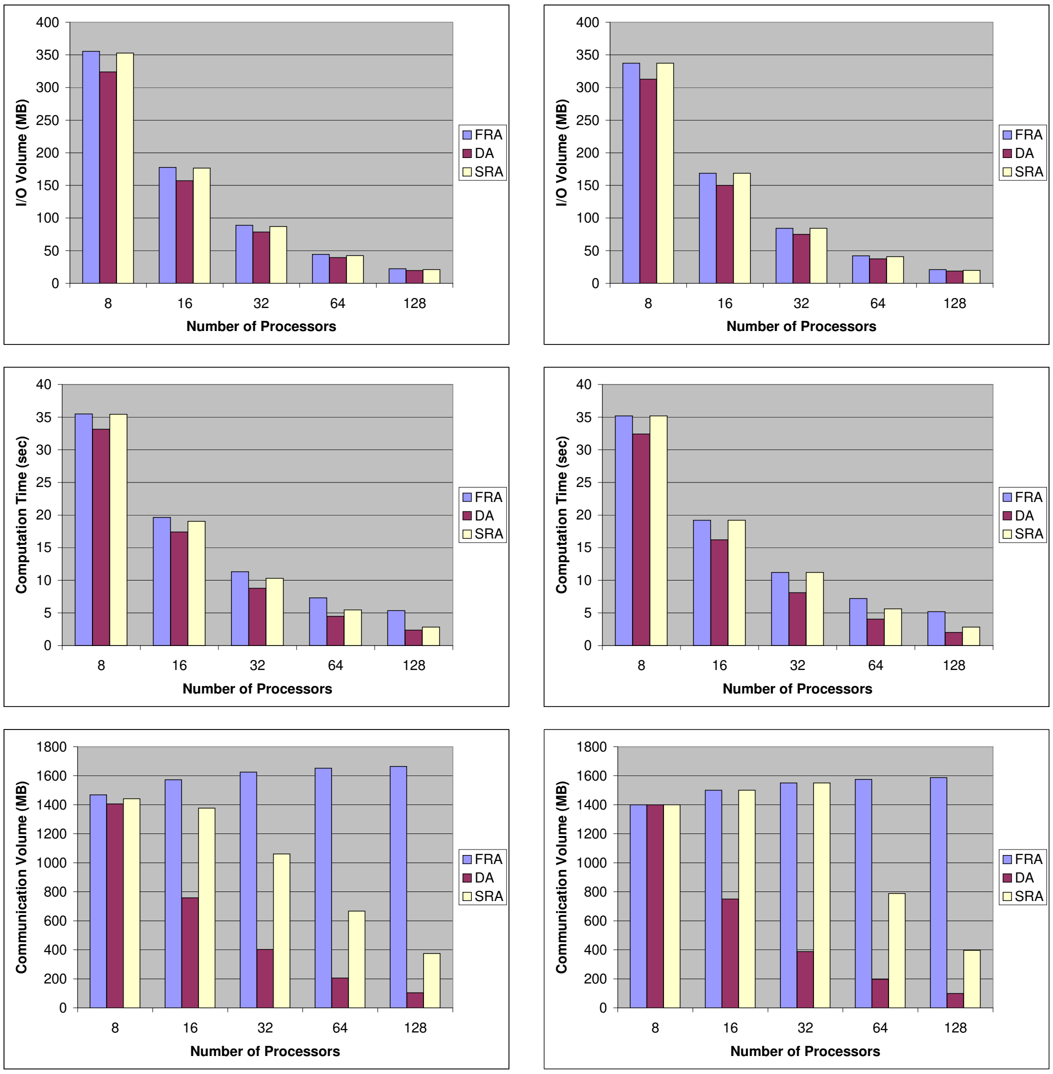



Figure 15: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = 4$, $\beta = 32$.

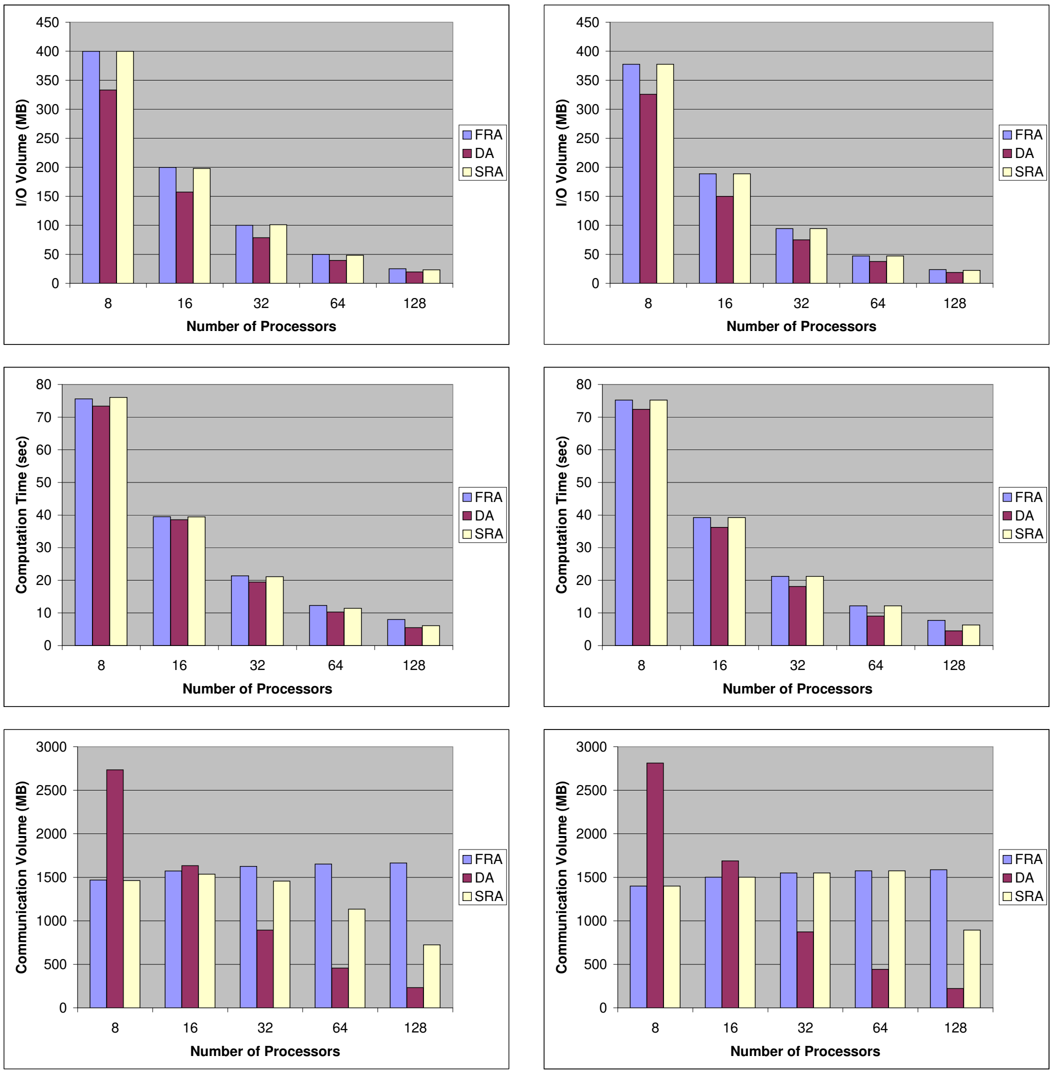



Figure 16: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = 9$, $\beta = 72$.

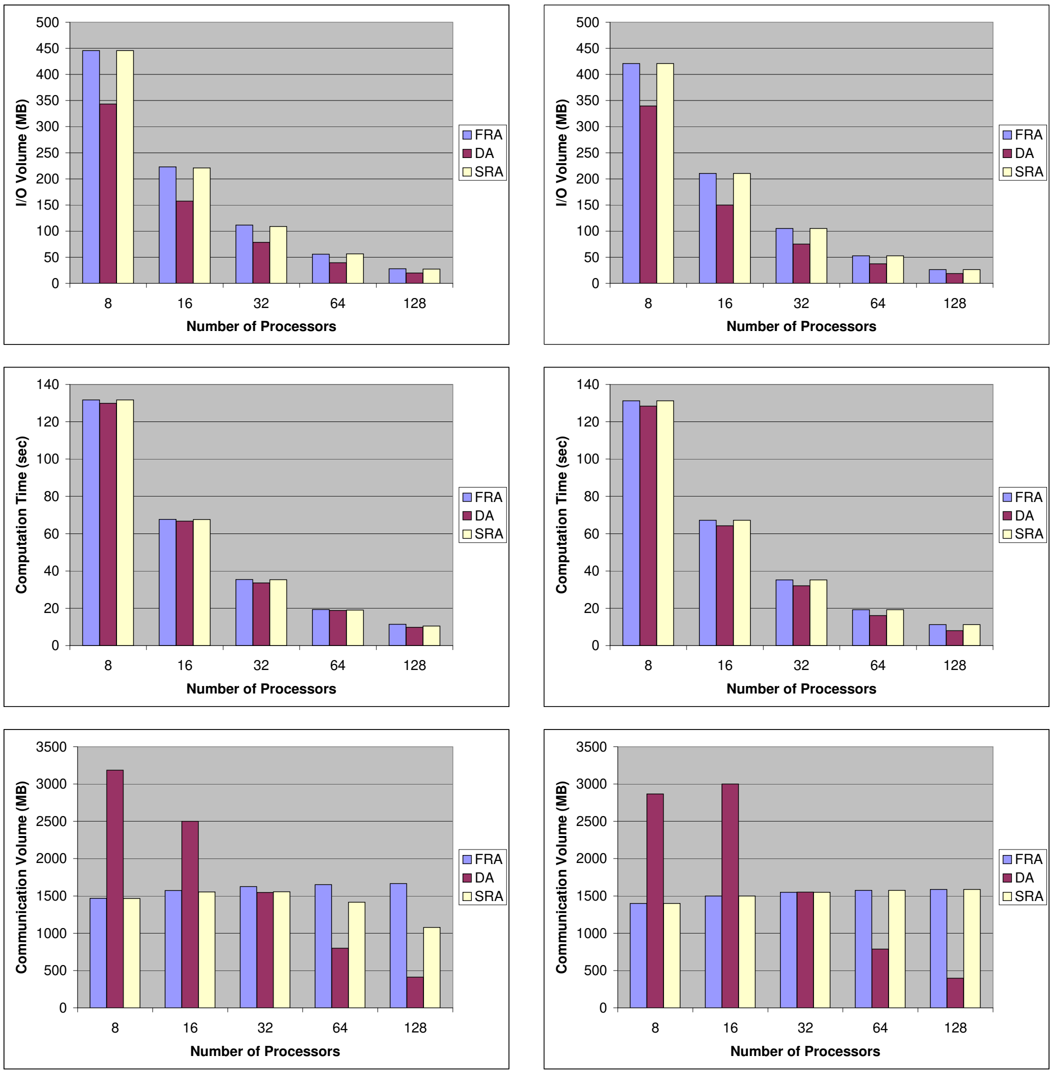



Figure 17: Actual (left) and estimated (right) I/O volume, computation time and communication volume for the synthetic query with $\alpha = 16$, $\beta = 128$.

DA. SRA often performs less computation than FRA because SRA may replicate fewer ghost chunks and therefore incur less computation overhead. This is confirmed by the observation that whenever SRA incurs less computation overhead than FRA , SRA also generates less communication volume. In fact, the cost model predicts that SRA behaves exactly the same as FRA until the point where $\beta = P$, after which an output chunk is only mapped to by $\beta$ input chunks, and therefore is only replicated by SRA on at most $\beta$ processors. FRA, on the other hand, always replicates an output chunk on all $P$ processors, and therefore generates more communication volume as the number of processors increases. The predicted relationship between FRA and SRA is most clearly displayed by the estimated communication volume in Figure 15. The actual communication volume shown in Figure 15, however, indicates that the actual behaviour of SRA tends to depart from that of FRA even with $\beta < P$. This is because the cost model assumes perfect declustering of the input chunks that map to an output chunk, which in practice is not achieved. As a result, SRA replicates an output chunk on fewer than $\beta$ processors, and therefore generates less communication volume that what the cost model predicts. Note that due to the same reason, the cost model for DA does not accurately estimate the communication volume for 16 processors for the query with $\alpha = 16$ and $\beta = 128$, as seen in Figure 17. The cost model assumes perfect declustering of the output chunks that an input chunk maps to. Thus, with $\alpha = 16$, an input chunk on one processor is expected to be sent to fifteen other processors. In practice, however, perfect declustering is not achieved, and an input chunk is sent to fewer than fifteen processors. As a result, the actual communication volume is less than what the cost model predicts.

We have also evaluated the cost models for different application scenarios, varying the number of processors and the input dataset size. We used *application emulators* [11] to generate various application scenarios for the applications classes that motivated the design of ADR (see Section 1). An application emulator provides a parameterized model of an application class; adjusting the parameter values makes it possible to generate different application scenarios within the application class and scale applications in a controlled way. The assignment of both input and output chunks to the disks was done using a Hilbert curve based declustering algorithm [5].

Table 3 summarizes dataset sizes and application characteristics for three application classes; *satellite data processing* (SAT) [4], analysis of microscopy data with the *Virtual Microscope* (VM) [1], and *water contamination studies* (WCS) [8]. The output dataset size was a fixed size for each application. The last column shows the computation time per chunk for the different phases of query execution (see Section 2); I-LR-GC-OH represents the Initialization-Local Reduction-Global Combine-Output Handling phases. The computation times shown represent the relative computation cost of the different phases within and across the different applications. The LR value denotes the computation cost for each intersecting (input chunk, accumulator chunk) pair. Thus, an input chunk that maps to a larger number of accumulator chunks takes longer to process. In all of these applications the output datasets are regular arrays, hence each output dataset is divided into regular multi-dimensional rectangular regions. The distribution of the individual data items and the data chunks in the input dataset for SAT is irregular. This is because of the polar orbit of the satellite [10]; the data chunks near the poles are more elongated on the surface of the earth than those near the equator and there are more overlapping chunks near the poles. The input datasets for WCS and VM are regular dense arrays that are partitioned into equal-sized rectangular chunks. We selected the values for the various parameters to represent some typical scenarios for these application classes, based on our experience with the complete applications.

Figures 18–20 show the measured and estimated values for I/O volume, computation time and communication volume for each application. As is seen from the figures, the cost models are able to estimate the volumes of I/O and communication in most application scenarios. However, the cost models fail to estimate the computation times of the strategies for the SAT and WCS applications. Our experiments show that in these two applications there is a load imbalance in the computation assigned to the various processors.

| App. | Input Dataset Num. of Chunks | Input Dataset Total Size | Output Dataset Num. of Chunks | Output Dataset Total Size | Average $\beta$ | Average $\alpha$ | Computation (in milliseconds) I–LR–GC–OH |
|---|---|---|---|---|---|---|---|
| SAT | 9K | 1.6GB | 256 | 25MB | 161 | 4.6 | 1–40–20–1 |
| WCS | 7.5K | 1.7GB | 150 | 17MB | 60 | 1.2 | 1–20–1–1 |
| VM | 16K | 1.5GB | 256 | 192MB | 64 | 1.0 | 1–5–1–1 |

Table 3: Application characteristics.

There are two main reasons for the load imbalance in these applications. First, the distribution of data elements in the output attribute space is not uniform for SAT. Second, the Hilbert curve-based declustering algorithms do not achieve optimal distribution of the input and output chunks across the processors, causing load imbalance in some cases. Since the cost models assume perfect declustering and a uniform distribution of the computations across the processors, the models may fail to predict the relative computation times of the strategies in those cases.

# 6 Conclusion

We have presented cost models to estimate the average operation counts for three query processing strategies, FRA, SRA and DA, in ADR. These cost models allow us to estimate the average I/O volume, computation time and communication volume for each processor. We have also validated our cost models with queries for synthetic datasets and queries for three driving applications. Our experiments show that our cost models are able to accurately estimate I/O volume, the computation time and communication volume for each of the three query processing strategies.

However, the ultimate goal of our research is to predict the relative query execution time for each query processing strategy so that for a given query and machine configuration, the ADR planning service is able to choose the query processing strategy that would process the query in the least amount of time. The cost models presented in this paper are able to accurately predict the I/O volume, computation time and communication volume for the three query processing strategies, but stop short of estimating the actual execution time. One solution is use the I/O and communication bandwidths of the machine that ADR runs on to turn I/O and communication volumes into I/O and communication times, and the average computation time for the data processing functions specified by the given query to turn computation operation counts into computation time. The sum can then be used as the estimated execution time. I/O and communication bandwidths can be measured by running a set of sample queries and use the average bandwidths observed by those queries. Computation time of the data processing functions can be obtained from statistics gathered either during ADR startup time or from earlier queries that invoke the same query processing functions. We are currently working on a machine model that when combined with the cost models presented in this paper, can be used to estimate the relative query executing times of the three strategies.

Note that the cost models that we have presented in this paper assume that the input data chunks are uniformly distributed in the output attribute space, and that the output chunks form a regular multi-dimensional grid. In scenarios where these assumptions do not hold, such as the SAT application described in Section 5, an inspector code can be used to generate a partial query plan for each of the three strategies and estimate the relative query execution times of the strategies based on information gathered from the partial query plans. The full query plan for the strategy that is predicted with the smallest execution time is then generated, and the query can be processed as planned. We are currently evaluating the effectiveness of

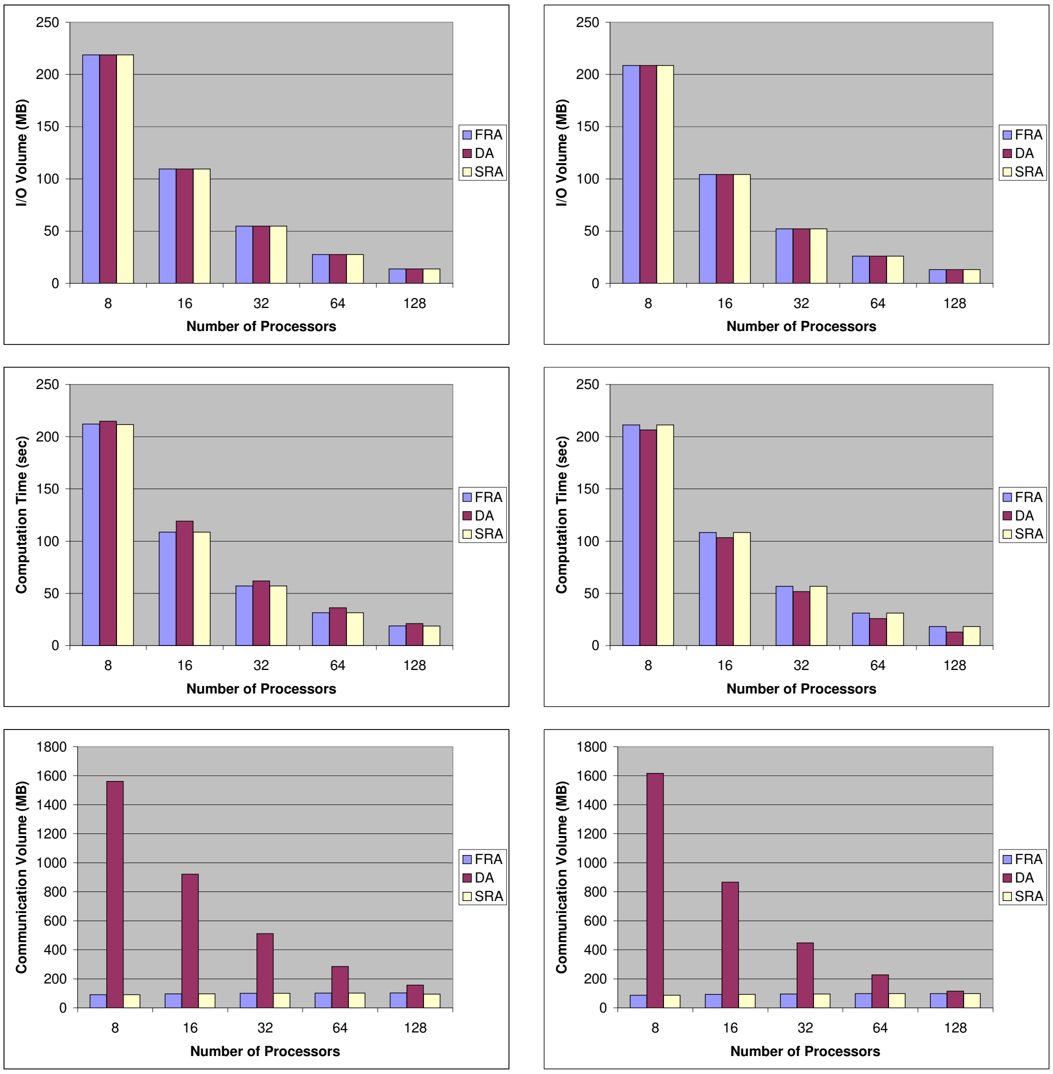



Figure 18: Measured (left) and estimated (right) I/O volume, computation time and communication volume for SAT application.

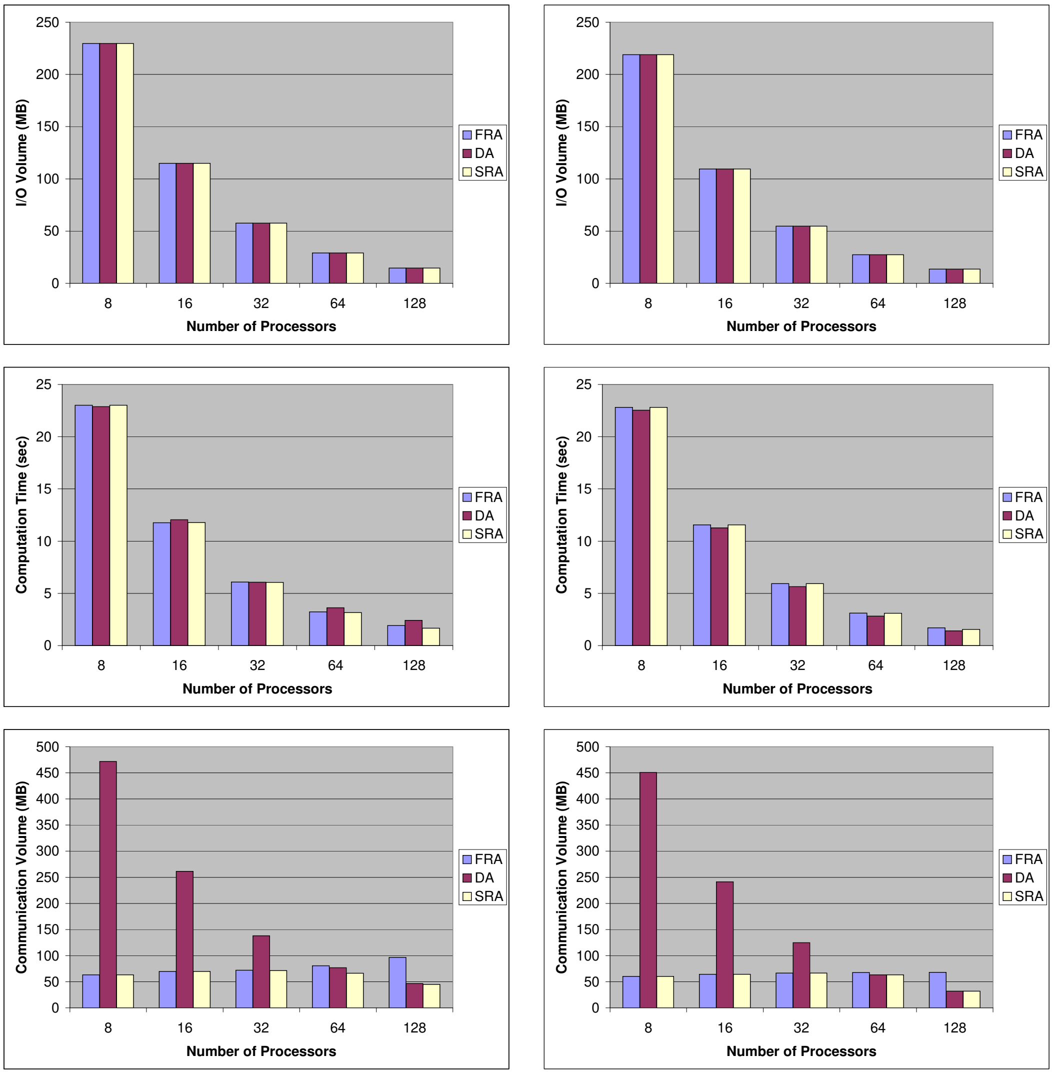



Figure 19: Measured (left) and estimated (right) I/O volume, computation time and communication volume for WCS application.

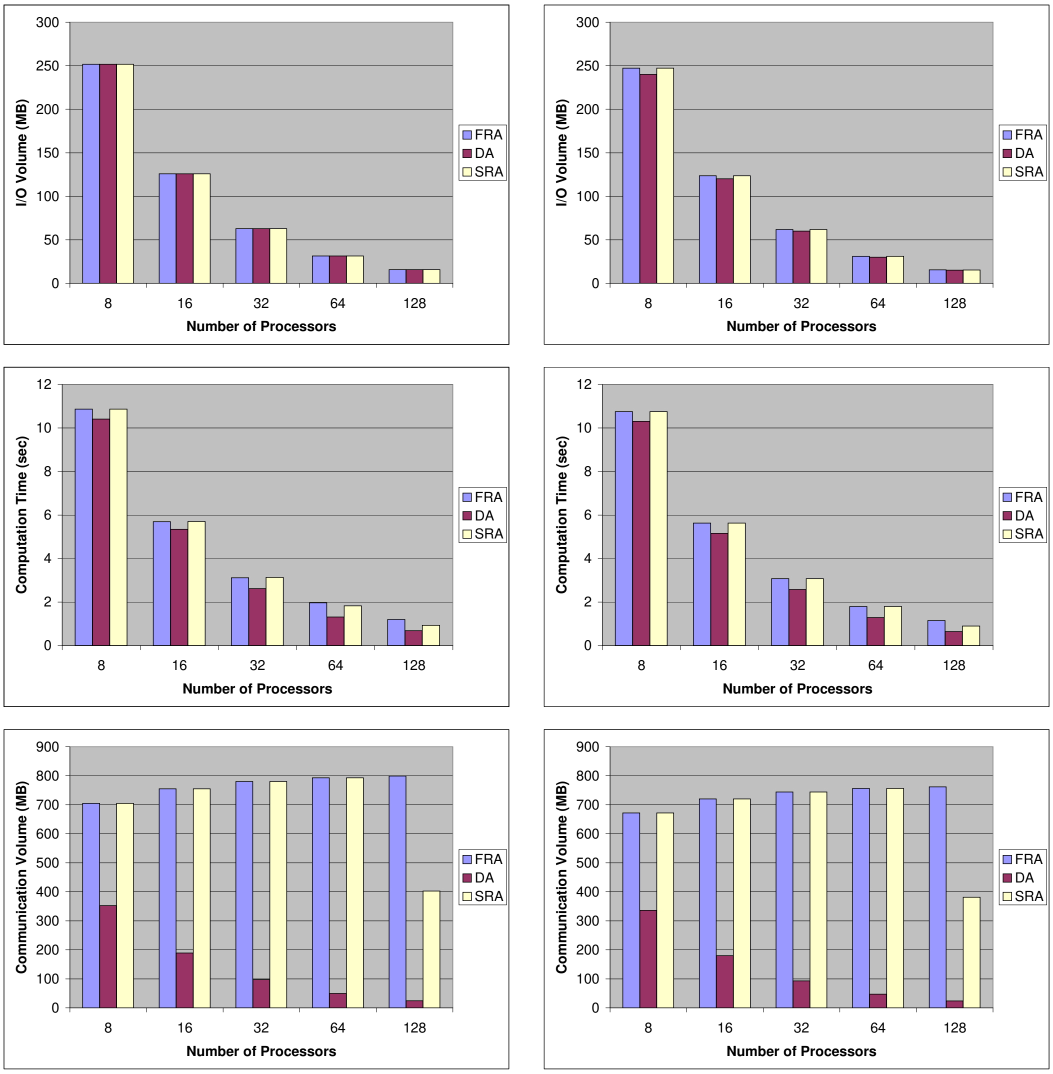



Figure 20: Measured (left) and estimated (right) I/O volume, computation time and communication volume for VM application.

such an inspector.

## References


[1] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, Nov. 1998.

[2] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, Apr. 1999.

[3] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Query planning for range queries with user-defined aggregation on multi-dimensional scientific datasets. Technical Report CS-TR-3996 and UMIACS-TR-99-15, University of Maryland, Department of Computer Science and UMIACS, Feb. 1999.

[4] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, Apr. 1997.

[5] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proccedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, San Diego, CA, Jan. 1993.

[6] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[7] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. Technical Report CS-TR-4022 and UMIACS-TR-99-29, University of Maryland, Department of Computer Science and UMIACS, May 1999. To appear in SC'99.

[8] T. M. Kurc, A. Sussman, and J. Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 1999.

[9] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.

[10] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. Available at http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html.

[11] M. Uysal, T. M. Kurc, A. Sussman, and J. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proceedings of the Fourth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in Lecture Notes in Computer Science, pages 243–258. Springer-Verlag, May 1998.