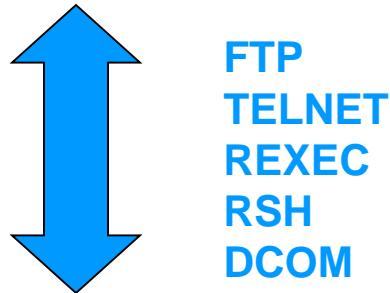


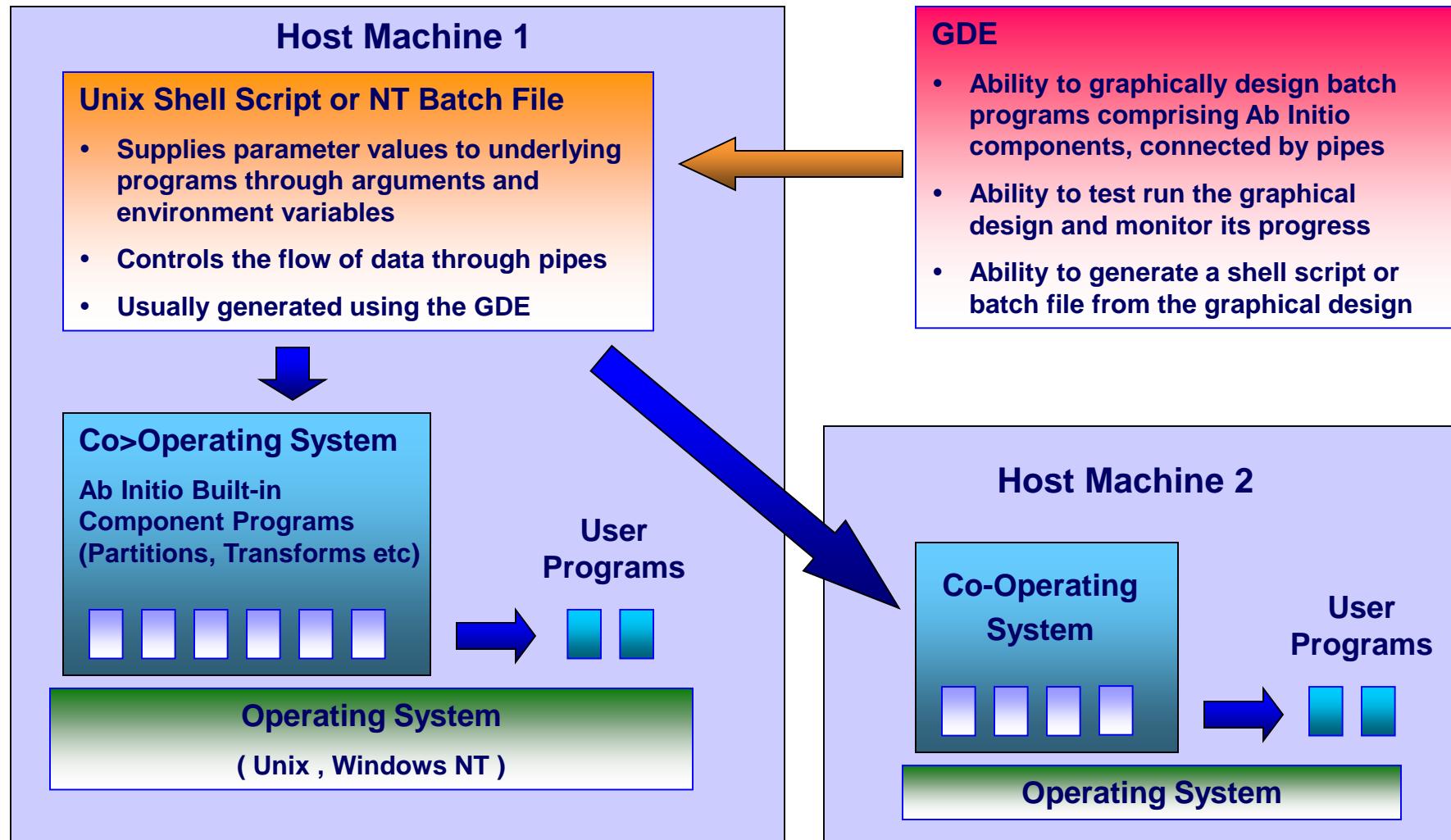
- * Data processing tool from Ab Initio software corporation
[\(http://www.abinitio.com\)](http://www.abinitio.com)
- * Latin for “from the beginning”
- * Designed to support largest and most complex business applications
- * Graphical, intuitive, and “fits the way your business works”text

Graphical Development Environment (GDE)



Co-operating System

On a typical installation, the Co-operating system is installed on a Unix or Windows NT server while the GDE is installed on a Pentium PC.



- * Layered on the top of the operating system
- * Unites a network of computing resources – CPUs, storage disks, programs, datasets into a data-processing system with scalable performance
- * Co>Operating system runs on ...

*IBM	AIX 4.1,4.2,4.3....
*Sun	Solaris 2.5.1,2.6,2.7....
*HP	HPUX 10, 11....
*Sequent	DYNIX/ptx 4.4.1, 4.4.2....
*Alpha	Digital Unix 4.0D, 4.4.2....
*Pyramid	Reliant Unix 5.43
*Intel	Windows NT 4.0

Co>Operating System Services ...

- * Parallel and distributed application execution
 - ✧ Control
 - ✧ Data Transport
- * Transactional semantics at the application level
- * Checkpointing
- * Monitoring and Debugging
- * Parallel file management
- * Metadata-driven components

The GDE ...

- * can talk to the Co-operating system using several protocols like Telnet, Rexec and FTP
- * is GUI for building applications in Ab Initio
- * and the Co-operating system have different release mechanisms, making Co-operating system upgradation possible without change in the GDE release

Note: During deployment, GDE sets AB_COMPATIBILITY to the Co>Operating System version number. So, a change in the Co>Operating System release may require a re-deployment

A Graph

- * is the logical modular unit of an application.
- * consists of several components that forms the building blocks of an Ab Initio application

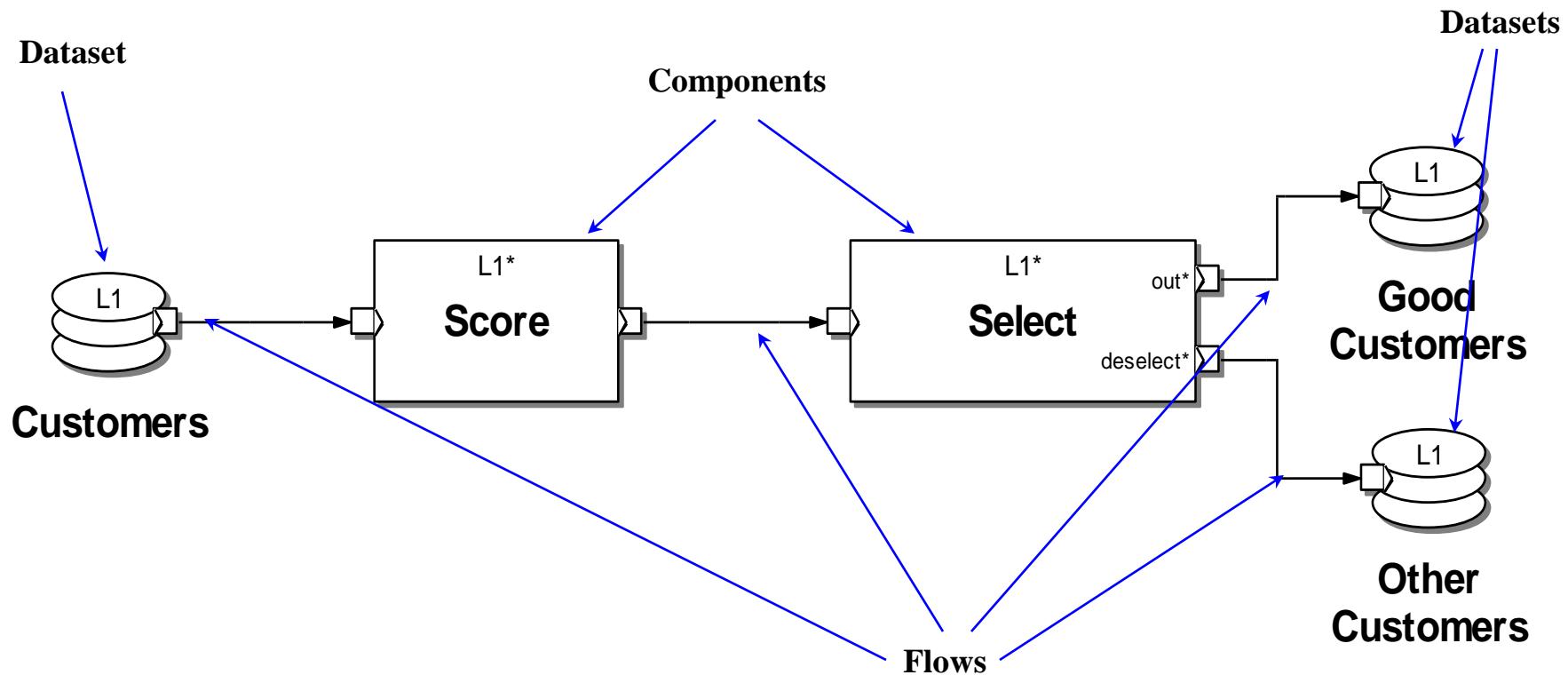
A Component

- * is a program that does a specific type of job and can be controlled by its parameter settings.

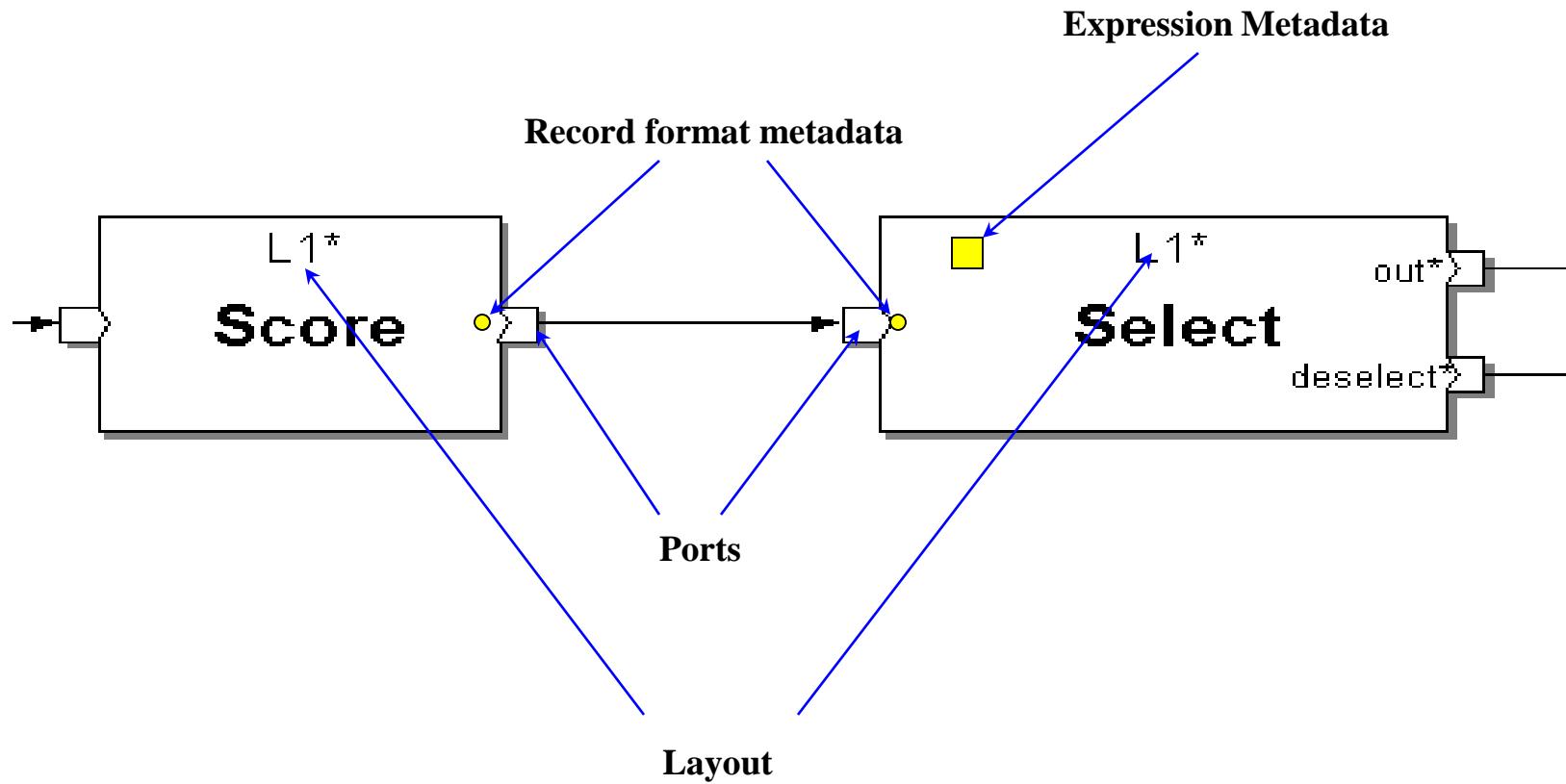
A Component Organizer

- * groups all components under different categories.

A Sample Graph ...



A Sample Graph ...



- * Files
- * Formats
- * Components
- * Flows
- * Layouts
- * Building with *mp job*
- * Building with *mp run*

* Setup Command

- ✧ **Ab Initio Host (AIH) file**
- ✧ **Builds up the environment to run an Ab Initio application.**

* Start Script (Host Setup)

- ✧ **Local to the Graph**

* Graph

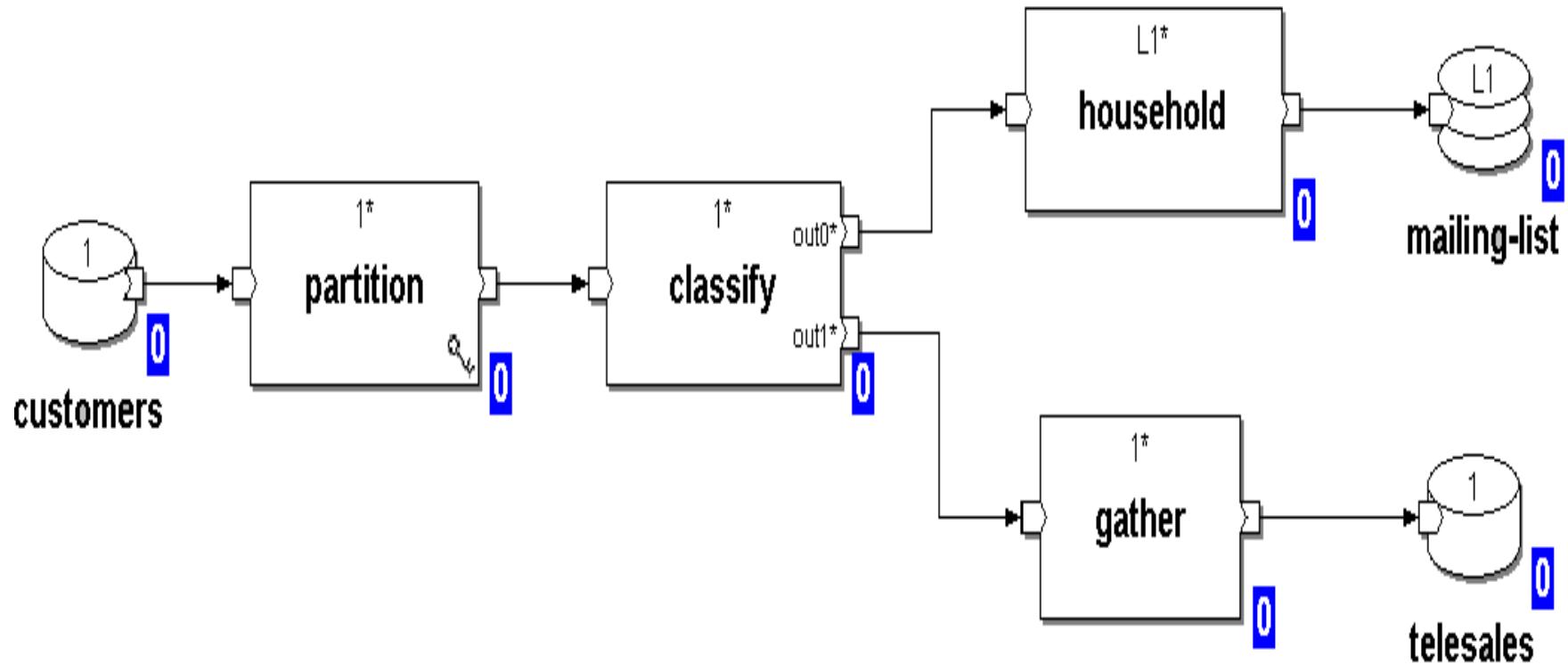
* End Script

- ✧ **Local to the Graph**

- * A graph, after development, is deployed to the back-end server as a Unix shell script or Windows NT batch file.
- * This becomes the executable to run at the back-end with the help of the Co-operating system.
- * The execution can be done from the GDE itself or manually from the back-end
- * Ab Initio runtime environment is different from the development environment.

A sample graph

Ab Initio



A sample korn shell script

An application script

Shown below is the entire script (`classify-customers.mp`) needed to implement the customer-classification program we have been examining:

```
#!/bin/ksh
set -e

mp job classify-customers

# Component      Name          Arguments
# -----        -----
mp ifile customers      file:customers
mp ofile telesales     file:telesales
mp ofile mailing-list   mfile:/mfs/mlist

mp hash-partition partition      "zipcode" -layout customers
mp classifier classify       -layout mailing-list
mp householder household     -layout mailing-list
mp gather gather         -layout telesales

# Define record format
mp metadata format           -file customers.dml

# Flow type      Name From Port      To Port      Data format
# -----        -----
mp straight-flow f1  customers.read  partition.in    -metadata format
mp fan-out-flow  f2  partition.out   classify.input  -metadata format
mp straight-flow f3  classify.mail   household.in   -metadata format
mp straight-flow f4  household.out   mailing-list.write -metadata format
mp straight-flow f5  classify.phone  gather.input   -metadata format
mp fan-in-flow   f6  gather.output   telesales.write -metadata format

# Run the Application
mp run
```

* DML

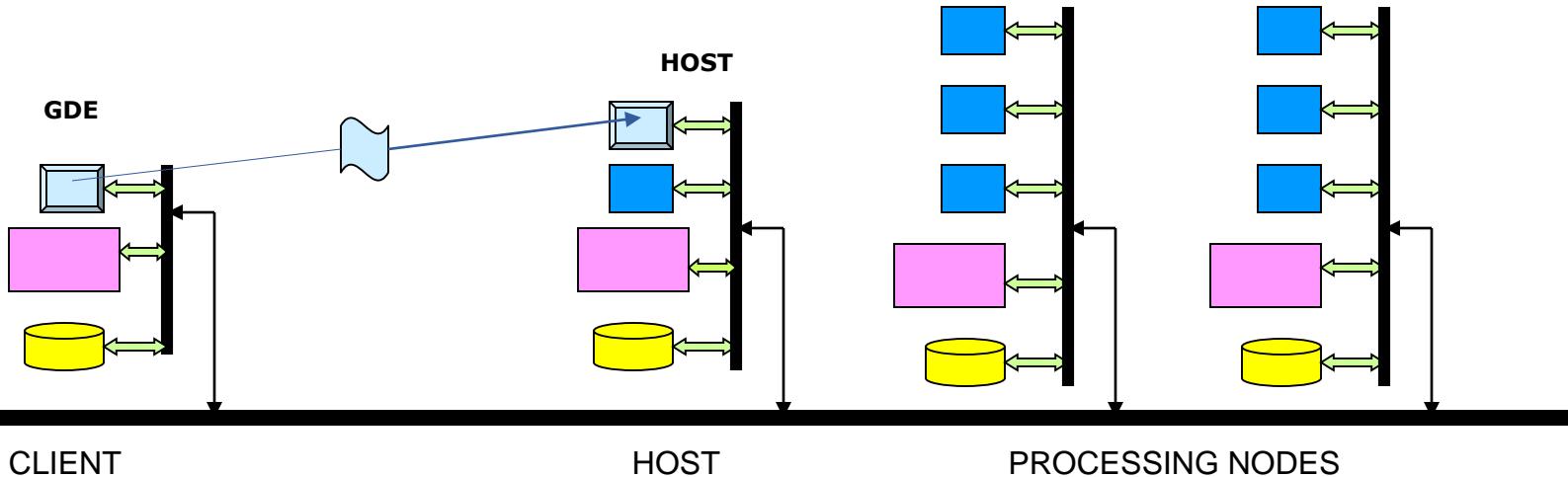
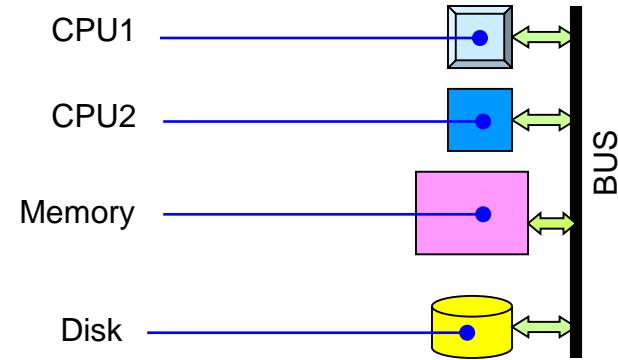
- ✧ Ab Initio stores metadata in the form of record formats.
- ✧ Metadata can be embedded within a component or can be stored external to the graph in a file with a “.dml” extension.

* XFR

- ✧ Data can be transformed with the help of transform functions.
- ✧ Transform functions can be embedded within a component or can be stored external to the graph in a file with a “.xfr” extension.

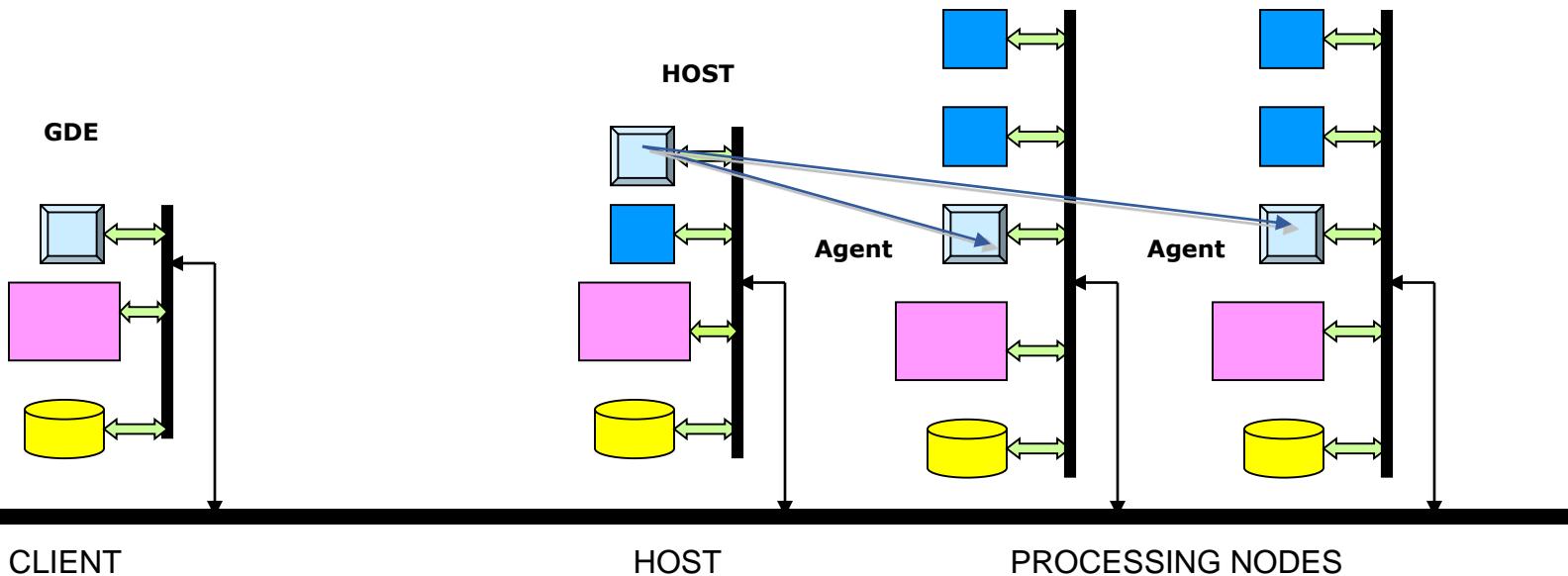
Host Process Creation

- * Pushing “Run” button generates script
- * Script is transmitted to Host node
- * Script is invoked, creating Host process



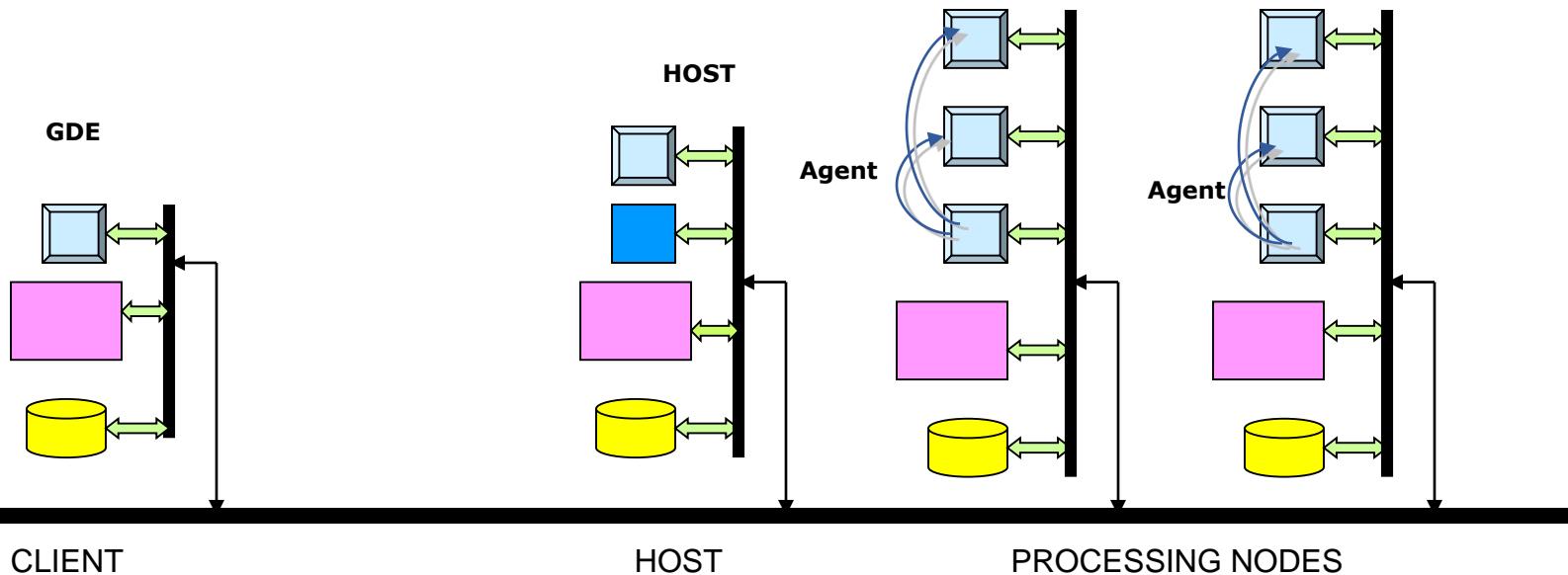
Agent Process Creation

- * Host process spawns Agent process



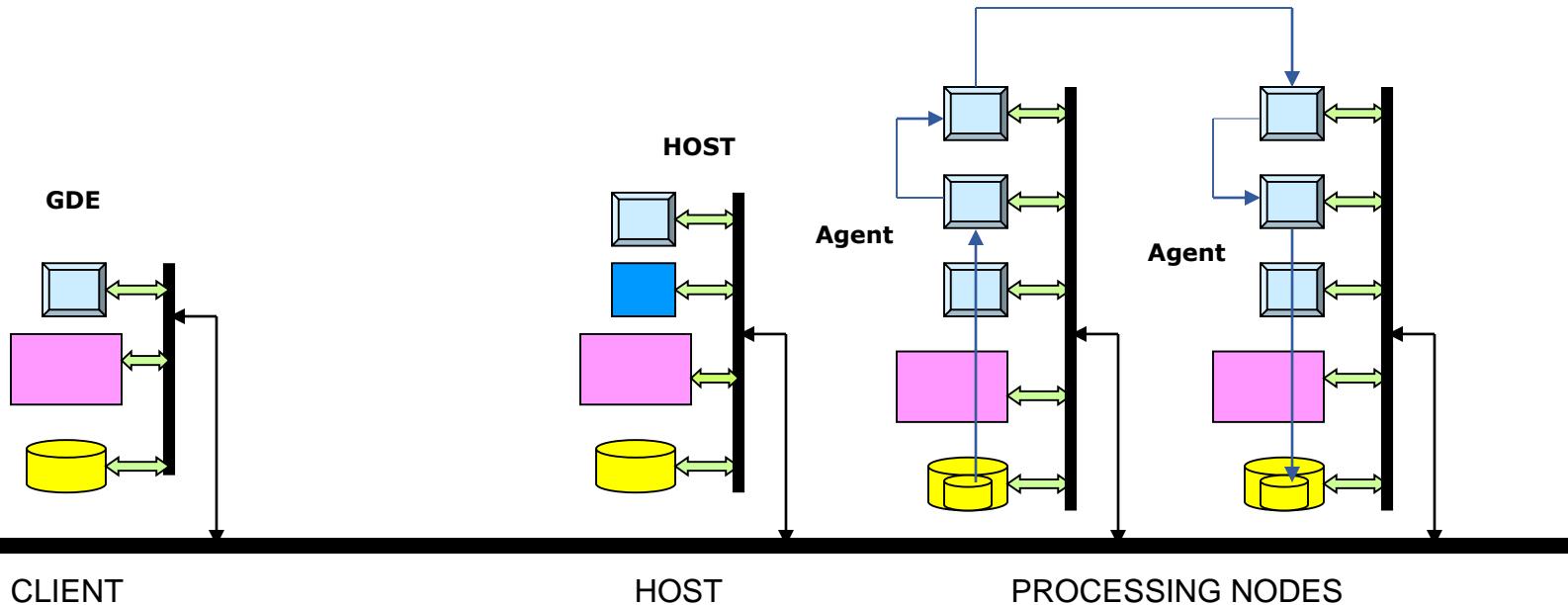
Component Process Creation

- * Agent process creates Component processes on each processing nodes



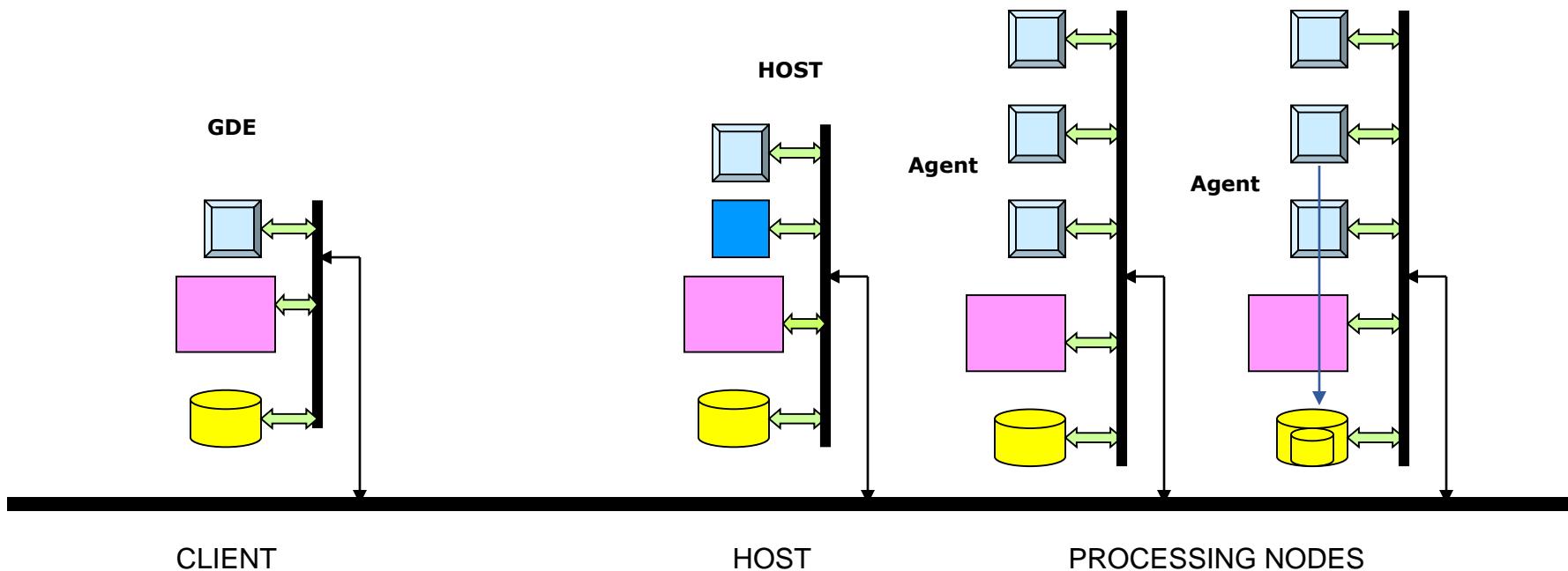
Component Process Execution

- * Component processes do their jobs.
- * Component processes communicate directly with datasets and each other to move data around



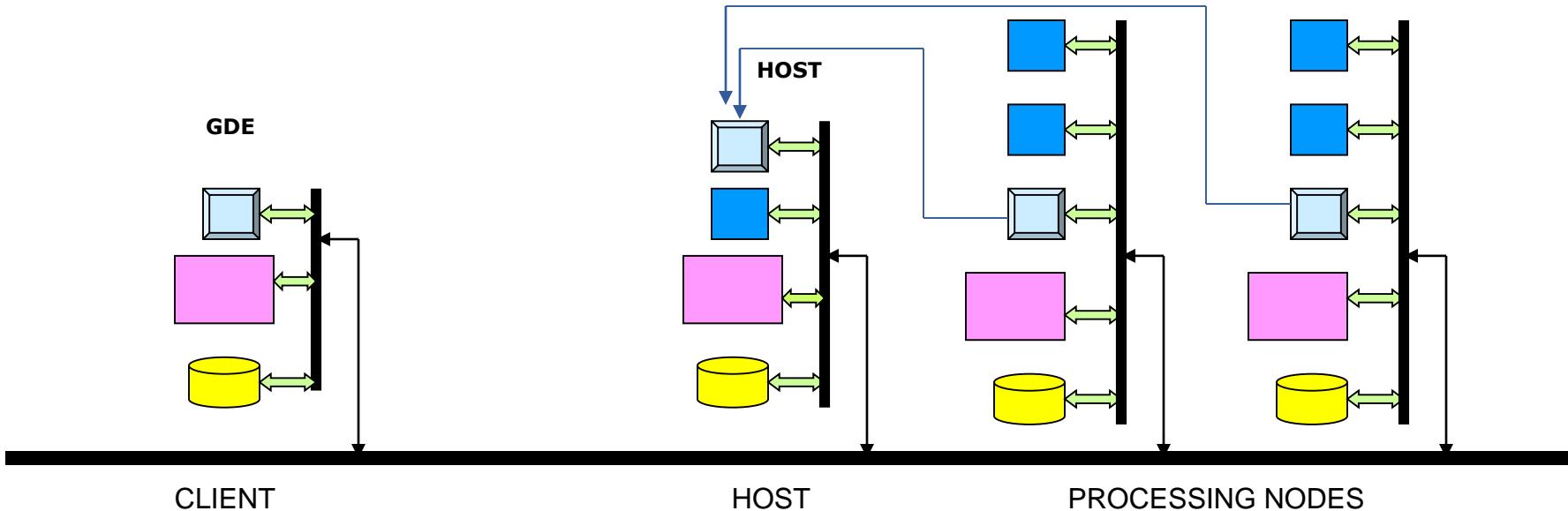
Successful Component Process Termination

- * As each Component process finishes with its data, it exits with success status



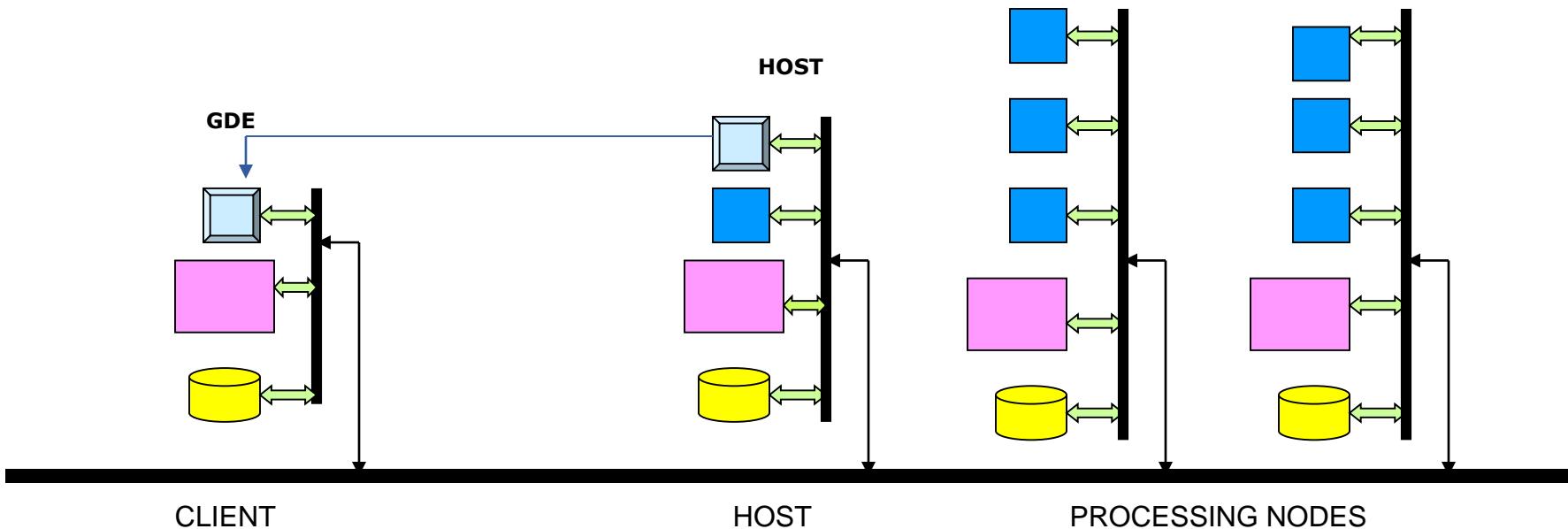
Successful Agent Process Termination

- * When all of an Agent's Component processes exit, the Agent informs the Host process that those component are finished.
- * The Agent process then exits.



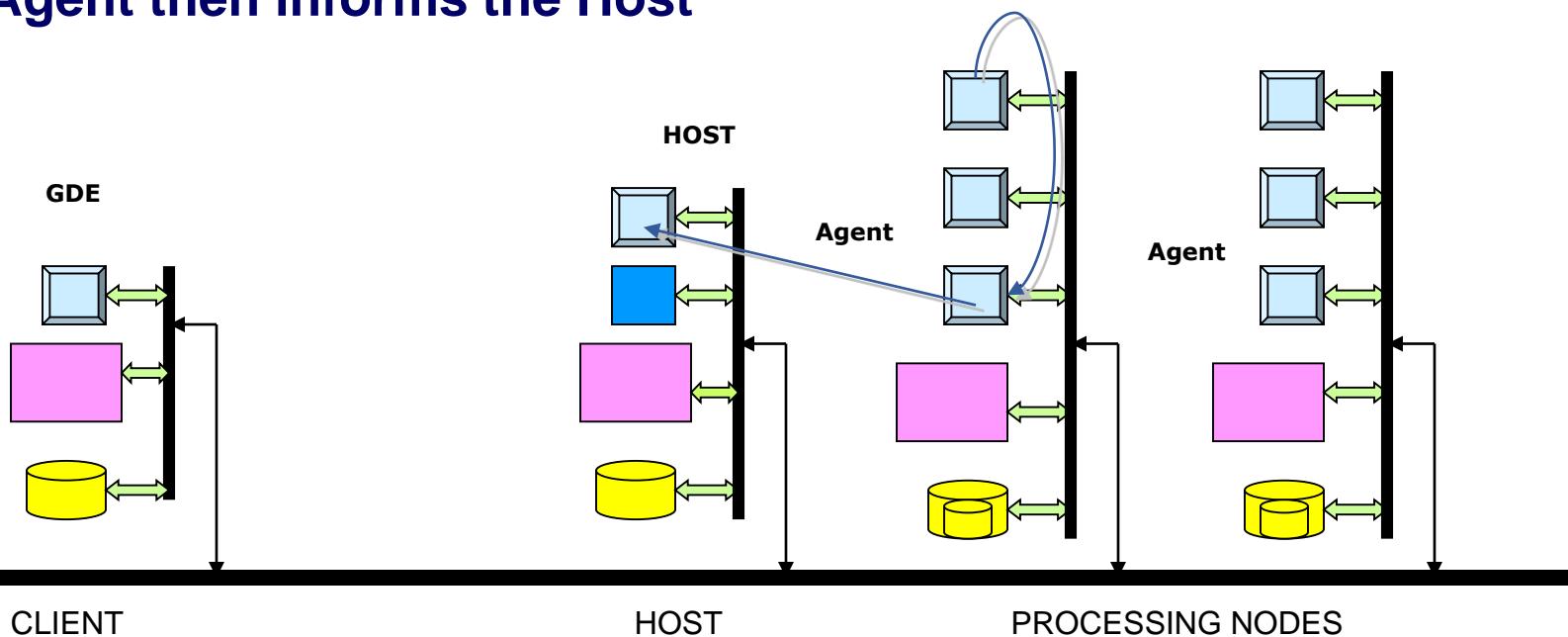
Successful Host Process Termination

- * When all have exited, the Host process informs the GDE that the job is complete.
- * The Host process then exits



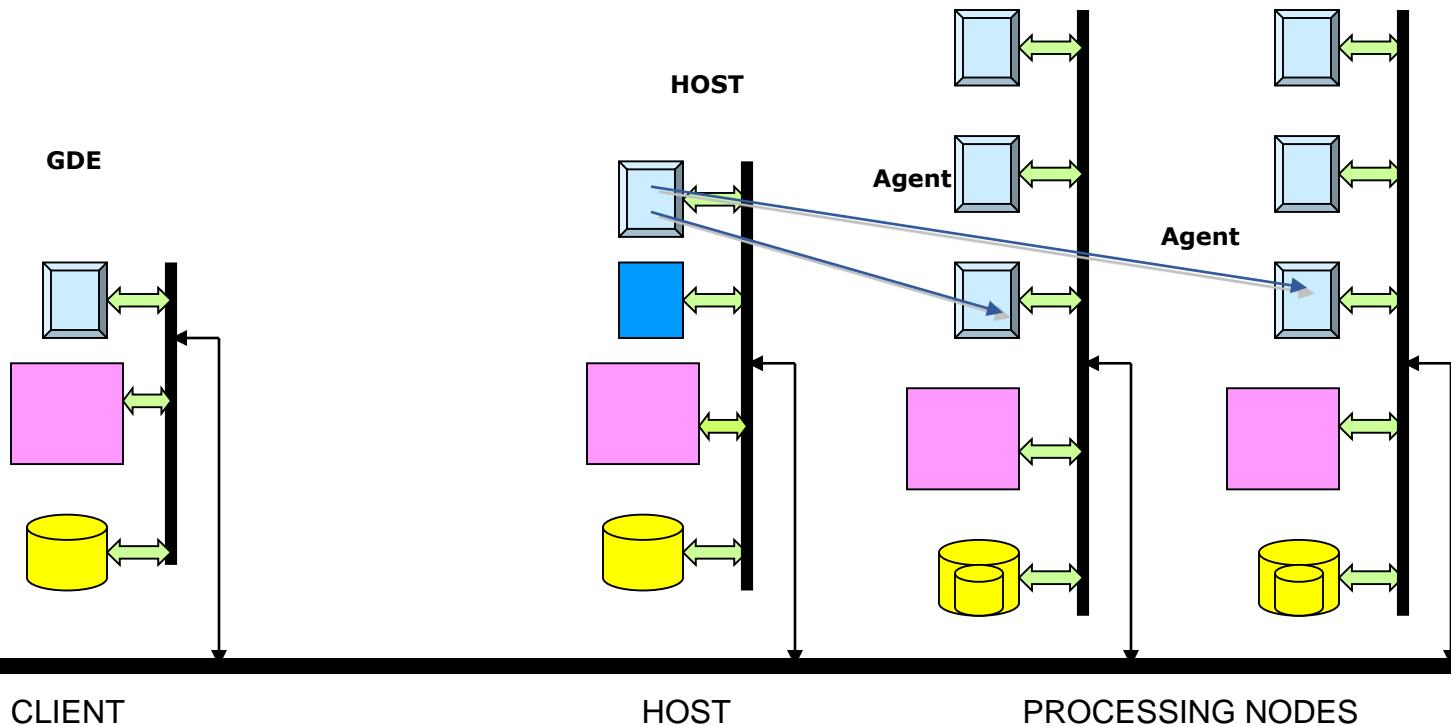
Abnormal Component Process Termination

- * When an error occurs in a Component process, it exits with error status.
- * The Agent then informs the Host



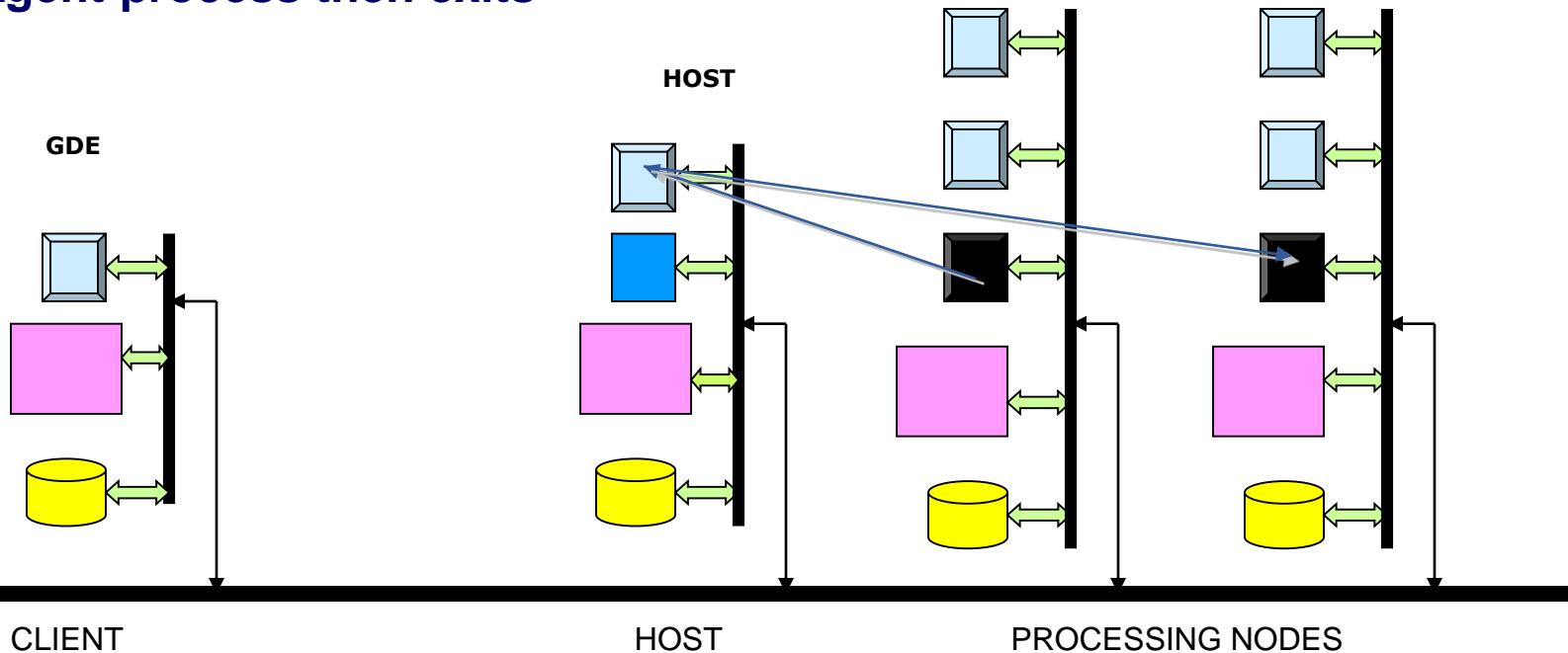
Abnormal Component Process Termination

- * The Host tells each Agent to kill its Component processes



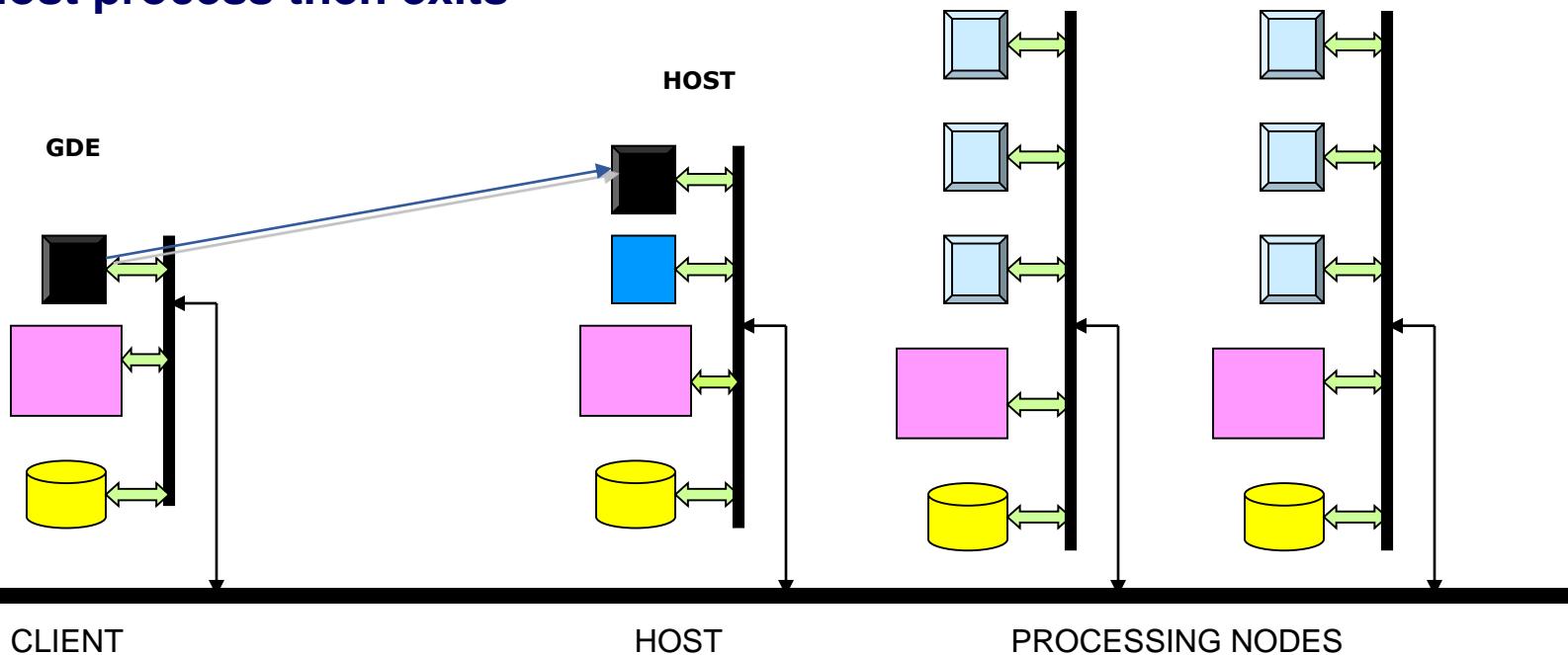
Agent Process Termination

- * When every Component process of an Agent have been killed, the Agent informs the Host process that those components are finished.
- * The Agent process then exits

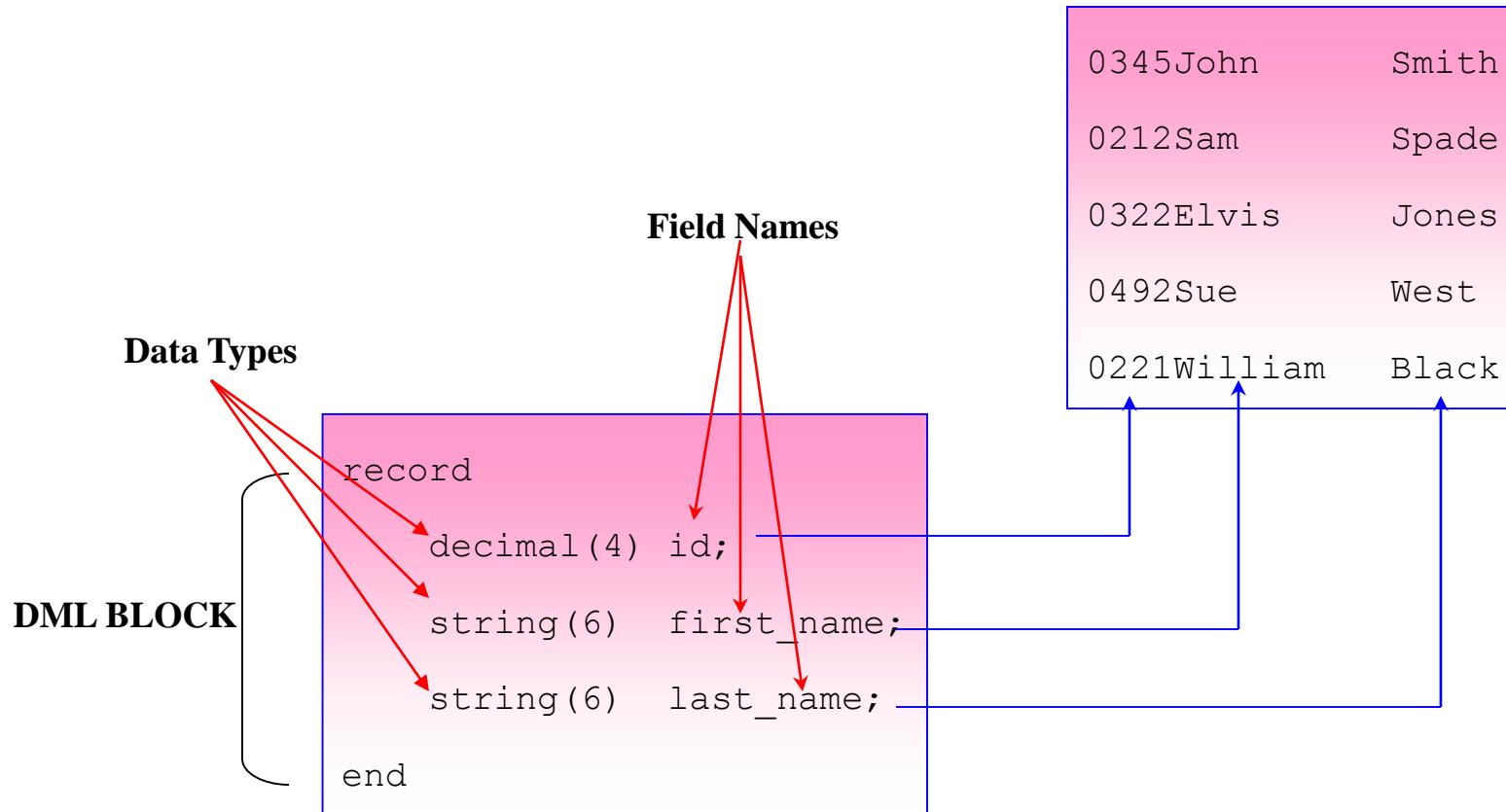


Host Process Termination

- * When all Agents have exited, the Host process informs the GDE that the job failed
- * The Host process then exits



Record Format Metadata in DML



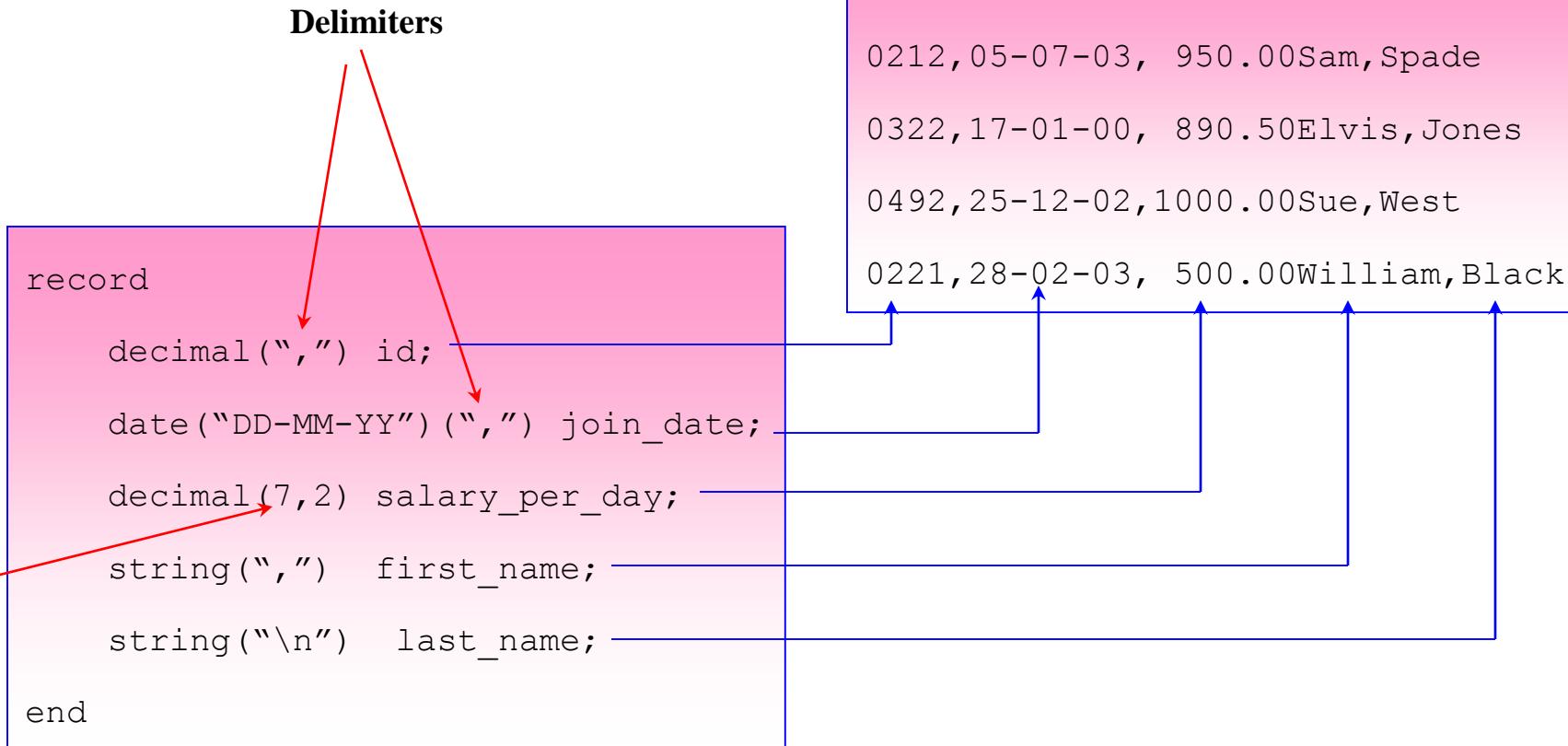
DML Syntax

- * Record types begin with ***record*** and end with ***end***
- * Fields are declared: ***data_type(len) field_name;***
- * Field names consist of letters(a...z,A...Z),digits(0...9) and underscores(_) and are ***Case sensitive***
- * Keywords/Reserved words are ***record, end, date....***

Data Types

- * String
- * Decimal
- * Integer
 - ✧ Storing Data in binary form
- * Date and Datetime
- * EBCDIC and ASCII records

More DML Types



Null in Oracle → Missing Information

Null in Ab Initio → Non-existence of columns

Ab Initio uses Conditional DML and “is_defined” function
in order to take care of NULL values.

Records

Header

Body

Trailer



Table Data

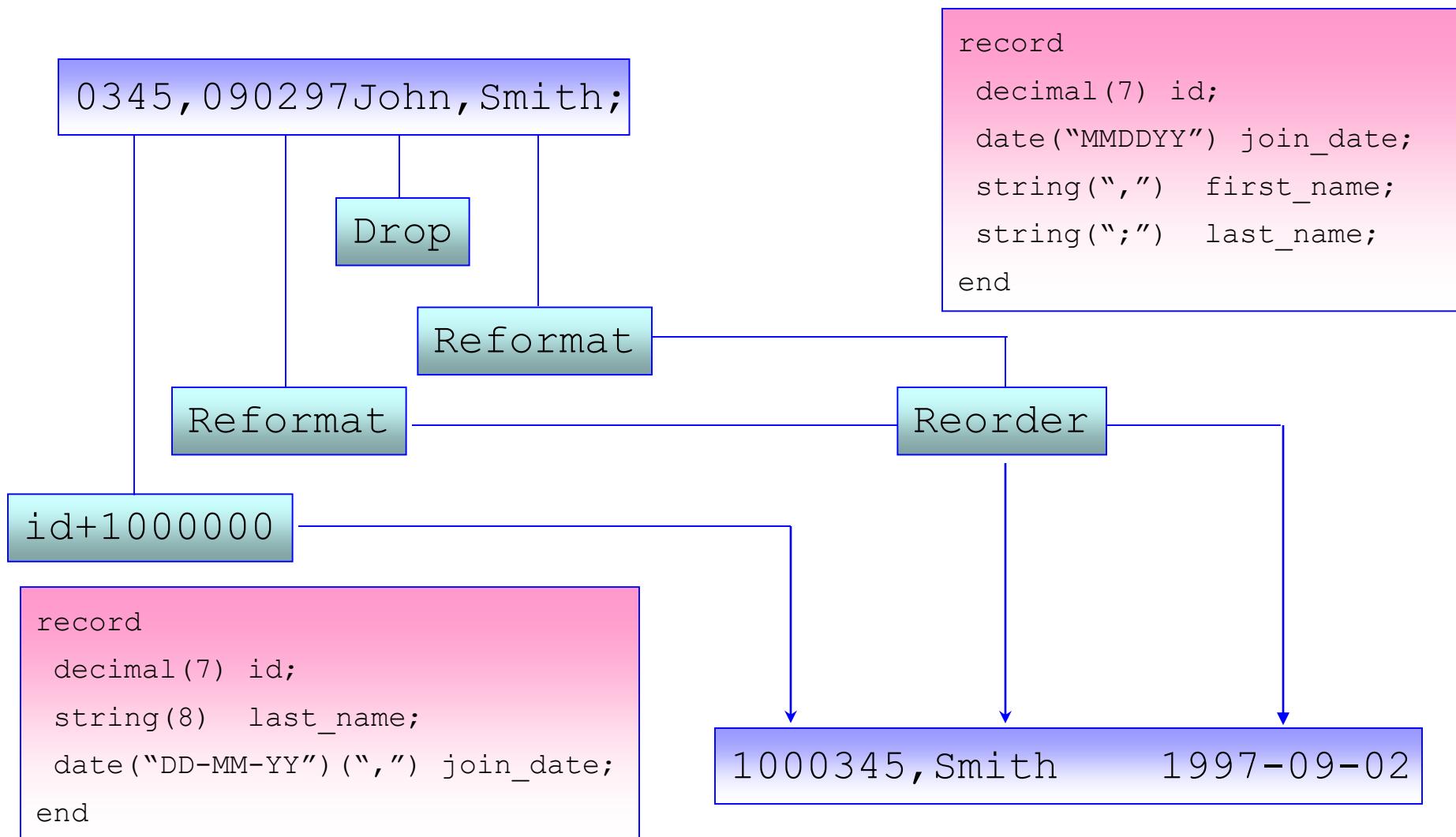
Header **NULL** **NULL**

NULL Body **NULL**

NULL **NULL** Trailer

Data Transformation

Ab Initio



- * User-defined function producing one or more output from one or more input
- * Associated with transform components
- * Rules that computes expression from input values and local variable and assigns the result to output objects
- * Syntax
 - ✧ Functions :

```
output-records :: function-name (input-records) =  
begin  
assignments  
end;
```
 - ✧ Assignments :

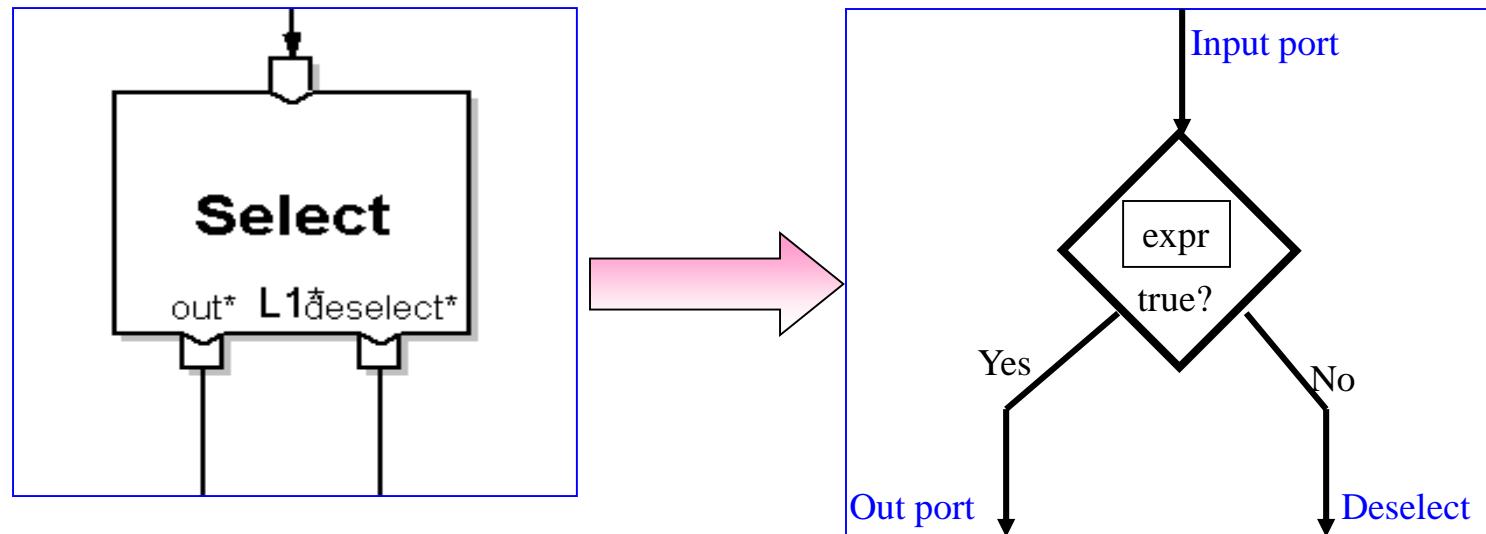
```
output-records.field :: expression;
```

Ab Initio >

DAY 2

- * Filter by Expression
- * Reformat
- * Redefine Format
- * Sort
- * Join
- * Replicate
- * Dedup
- * Aggregate
- * Rollup
- * Scan

1. Reads record from *input* port
2. Evaluate the *select_expr*
3. If result is true, record written to *out* port
4. If result is false, record written to *deselect* port



* REJECT

- ✧ Input records that caused error

* ERROR

- ✧ Associated error message

* LOG

- ✧ Logging records

Note: Every transform component has got diagnostic ports

1. Reads record from *input* port
2. Record passes as argument to *transform function or xfr*
3. Records written to *out* ports, if the function returns a success status
4. Records written to *reject* ports, if the function returns a failure status

Parameters of Reformat Component

- * **Count**
- * **Transform (Xfr) Function**
- * **Reject-Threshold**
 - ✧ **Abort**
 - ✧ **Never Abort**
 - ✧ **Use Limit & Ramp**
 - ⇒ **Limit**
 - ⇒ **Ramp**

* Limit

- ◇ Number of errors to tolerate

* Ramp

- ◇ Scale of errors to tolerate per input

* Tolerance value=limit + ramp*total number of records read

* Typical Limit and Ramp settings . .

- ◇ Limit = 0 Ramp = 0.0 → *Abort on any error*
- ◇ Limit = 50 Ramp = 0.0 → *Abort after 50 errors*
- ◇ Limit = 1 Ramp = 0.01 → *Abort if more than 1 in 100 records causes error*
- ◇ Limit = 1 Ramp = 1 → *Never Abort*

Keys

A key identifies a field or set of fields to organize a dataset

- ✧ Single Field: *employee_number*
- ✧ Multiple field or Composite key: (*last_name; first_name*)
- ✧ Modifiers: *employee_number descending*

A surrogate key is a substitution for the natural primary key.

- ⇒ It is just a unique identifier or number for each record like **ROWID** of an Oracle table

Sort Component

*Reads records from input port, sorts them by key, writes result to output port

*Parameters

- ✧ Key
- ✧ Max-core

1. Reads records from multiple input ports
2. Operates on records with matching keys using a multi-input transform function
3. Writes result to the output port

PORtS

- in
- out
- unused
- reject (optional)
- error (optional)
- log (optional)

PARAMETERS

- count
- key
- override key
- transform
- limit
- ramp

Join Types

- ✧ **Inner**
- ✧ **Outer**
- ✧ **Explicit**

Join Methods

- ✧ **Merge Join**
 - ⇒ **Using sorted inputs**
- ✧ **Hash Join**
 - ⇒ **Using in-memory hash tables to group input**

The **Priority** is the order of evaluation of rules in a transform function.

An example

A join component may have a transform function with prioritized rules as

```
out.ssn :1: in1.ssn;  
out.ssn :2: in2.ssn;  
out.ssn :3: "999999999";
```

- * Data transformation in multiple stages following several sets of rules
- * Each set of rule form one transform function
- * Information is passed across stages by temporary variables
- * Stages include initialization, iteration, finalization and more
- * Few multistage components are aggregate,rollup,scan

Aggregate/Rollup/Scan

Generates summary records for group of input records

Components contd..

Name	Description
Normalize	<ul style="list-style-type: none"> ★Generates multiple data records from each input data record ★Separate a data record with a vector field into several individual records, each containing one element of the vector.
Denormalize Sorted	<ul style="list-style-type: none"> ★Consolidates groups of related data records into a single output record with a vector field for each group ★Requires Grouped Input
Validate Records	Separates valid data records from invalid data records
Check Order	Tests whether data records are sorted according to a key-specifier.
Compare Records	Compares data records from two flows one by one
Generate Records	Generates a specified number of data records with fields of specified lengths and types.
Gather Logs	Collects the output from the log ports of components for analysis of a graph after execution
Sample	Selects a specified number of data records at random from one or multiple input flows

Ab Initio built-in functions are DML expressions that

- ✧ can manipulate strings, dates, and numbers
- ✧ access system properties

Function categories

- ✧ Date functions : `now()`, `today()`, `date_to_int()`, ..
- ✧ Inquiry and error functions: `is_defined()`, `is_valid()`, `force_error()`, ..
- ✧ Lookup functions: `lookup()`, `lookup_local()`, ..
- ✧ Math functions: `ceiling()`, `floor()`, ..
- ✧ Miscellaneous functions:`decimal_round()`, `hash_value()`, ..
- ✧ String functions: `string_substring()`, `is_blank()`, ..

*Perform pattern matching operations on strings using regular expressions

Name	Function	Example
re_get_match()	Returns the first string that matches a regular expression.	<ul style="list-style-type: none"> •<code>re_get_match("man on the moon", "oo")</code> → "oo" •<code>re_get_match("1234 Milldale Ave.", "[a-zA-Z]+")</code> → "Milldale" •<code>re_get_match("1234 Milldale Ave.", "[0-9]+")</code> → "1234" •<code>re_get_match("Fourteen 35th St.", "[0-9]+")</code> → <i>NULL</i>
re_index()	Returns the index of the first character of a string matching a regular expression.	<code>re_index("man on the moon", "oo")</code> → 13 <code>re_index("1234 Milldale Ave.", "[a-zA-Z]+")</code> → 6 <code>re_index("1234 Milldale Ave.", "[0-9]+")</code> → 1 <code>re_index("Fourteen 35th St.", "[0-9]+")</code> → 0
re_replace()	Returns a string after replacing all the substrings matching a regular expression	<code>re_replace("man on the moon", "m[aeiou]+n", "X")</code> → " X on X"
re_replace_first()	Returns a string after replacing the first substring matching a regular expression.	<code>re_replace_first("man on the moon", "m[aeiou]+n", "X")</code> → "X on the moon"

A Vector is

*an array of same type of elements that is repeated

*number of repeats is a constant integer or the value of another field in the record.

```
record  
    string(20) cust_id;  
  
    decimal(3) num_purchases;  
  
    decimal(8.2)[num_purchases]purchase_amt;  
end;
```

NOTE: purchase_amount field is repeated depending on the value in the number_of_purchases field

Initializing a Vector

```
let decimal(',')[100] values = make_constant_vector(100, 0);
```

Converting a string into a vector

```
let decimal(',')[3] values = reinterpret_as(val_str, ',', 3);
```

***db_config_utility** : Generate interface file to the database

*Input Table

- ✧ unloads data records from a database into an Ab Initio graph
- ✧ Source : DB table or SQL statement to SELECT from table

*Output Table

- ✧ loads data records into a database
- ✧ Destination : DB table or SQL statement to INSERT into table

*Update Table

- ✧ executes UPDATE or INSERT statements in embedded SQL format to modify a DB table

*Truncate Table

- ✧ deletes all the rows in a specified DB table

*Run SQL

- ✧ executes SQL statements in a DB

Ab Initio >

DAY 3

Parallel Runtime Environment

Where some or all of the components of an application – datasets and processing modules are replicated into a number of partitions, each spawning a process.

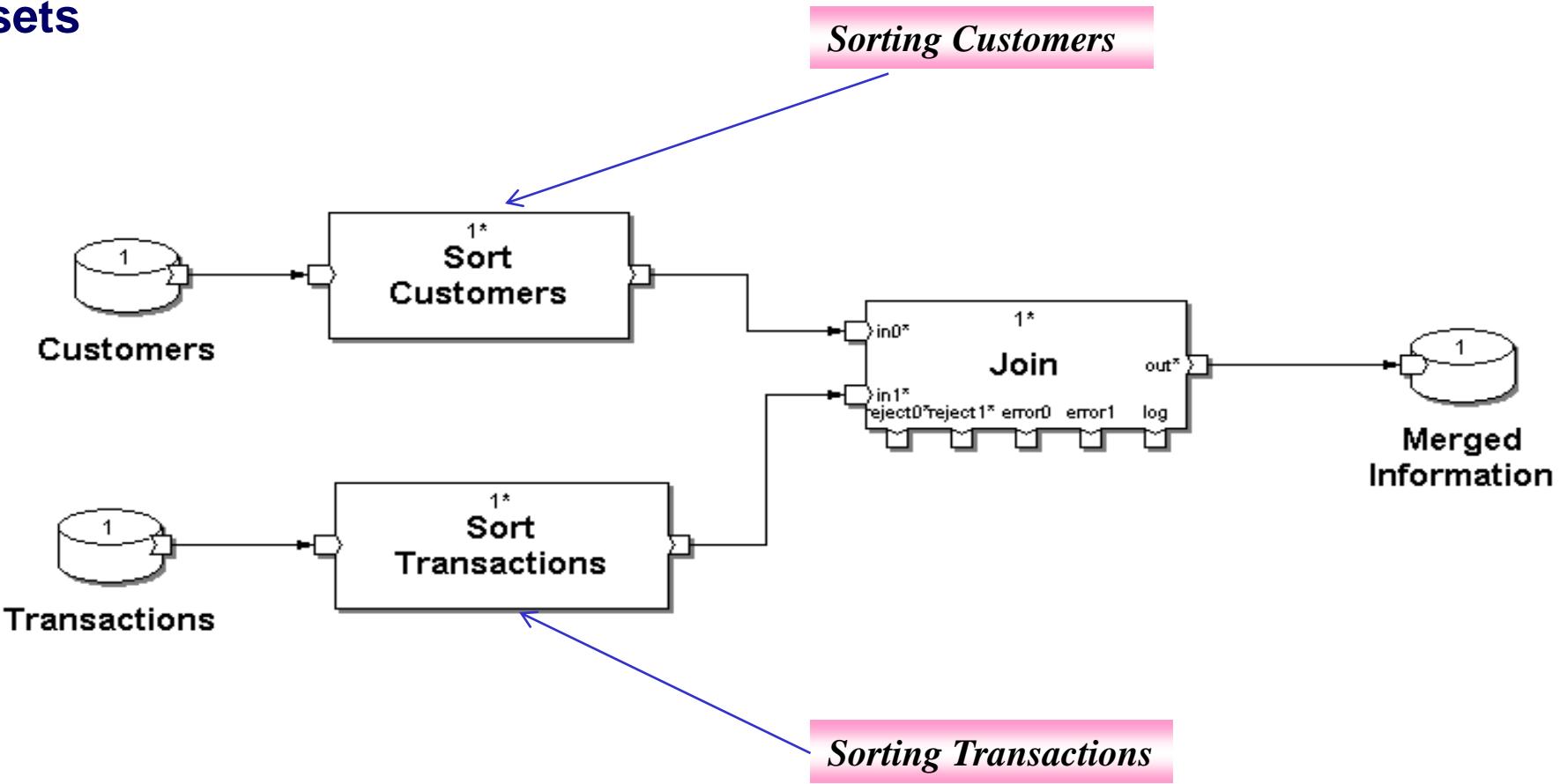
Ab Initio can process data in parallel runtime environment

Forms of Parallelism

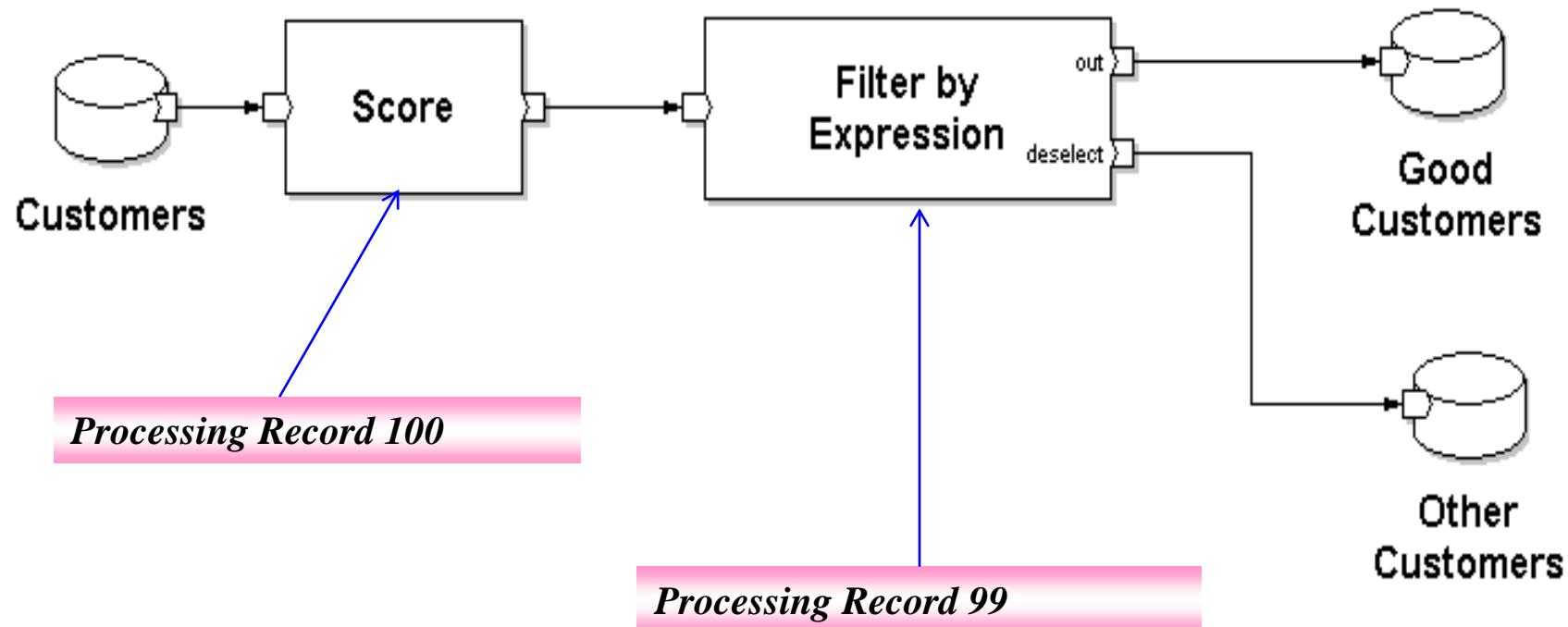
- ✧ Component Parallelism
- ✧ Pipeline Parallelism
- ✧ Data Parallelism

} *Inherent in Ab Initio*

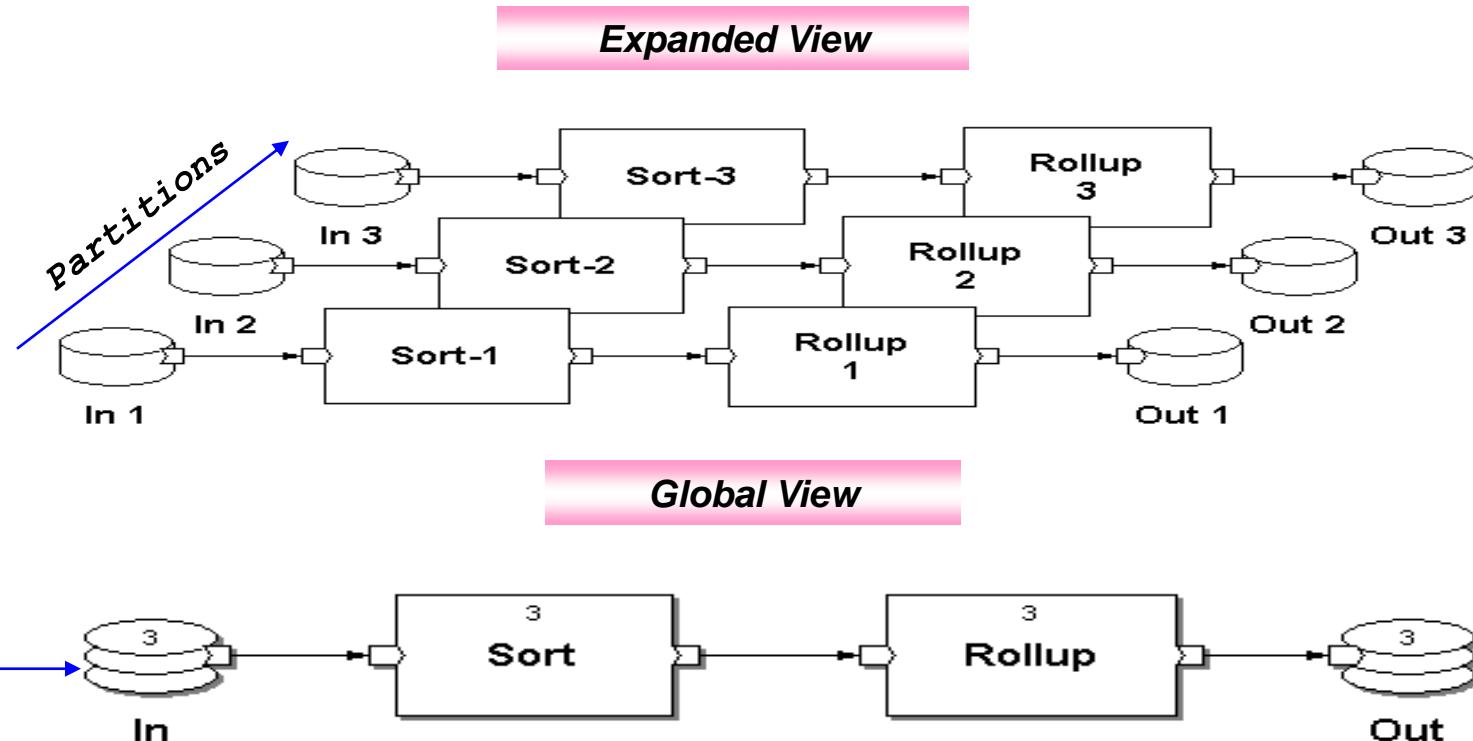
When different instances of same component run on separate data sets



When multiple components run on same data set



When data is divided into segments or *partitions* and processes run simultaneously on each *partition*

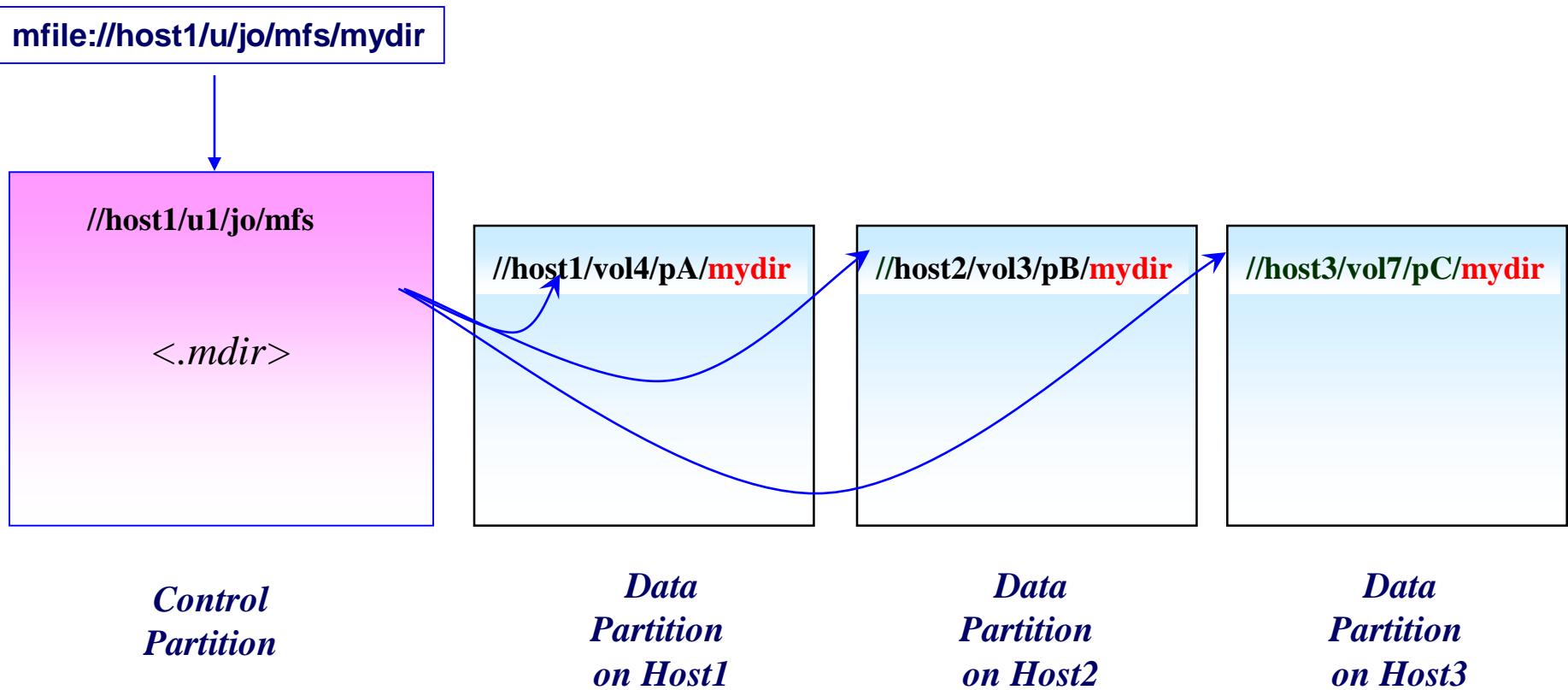


NOTE : # of processes per component = # of partitions

Multifiles

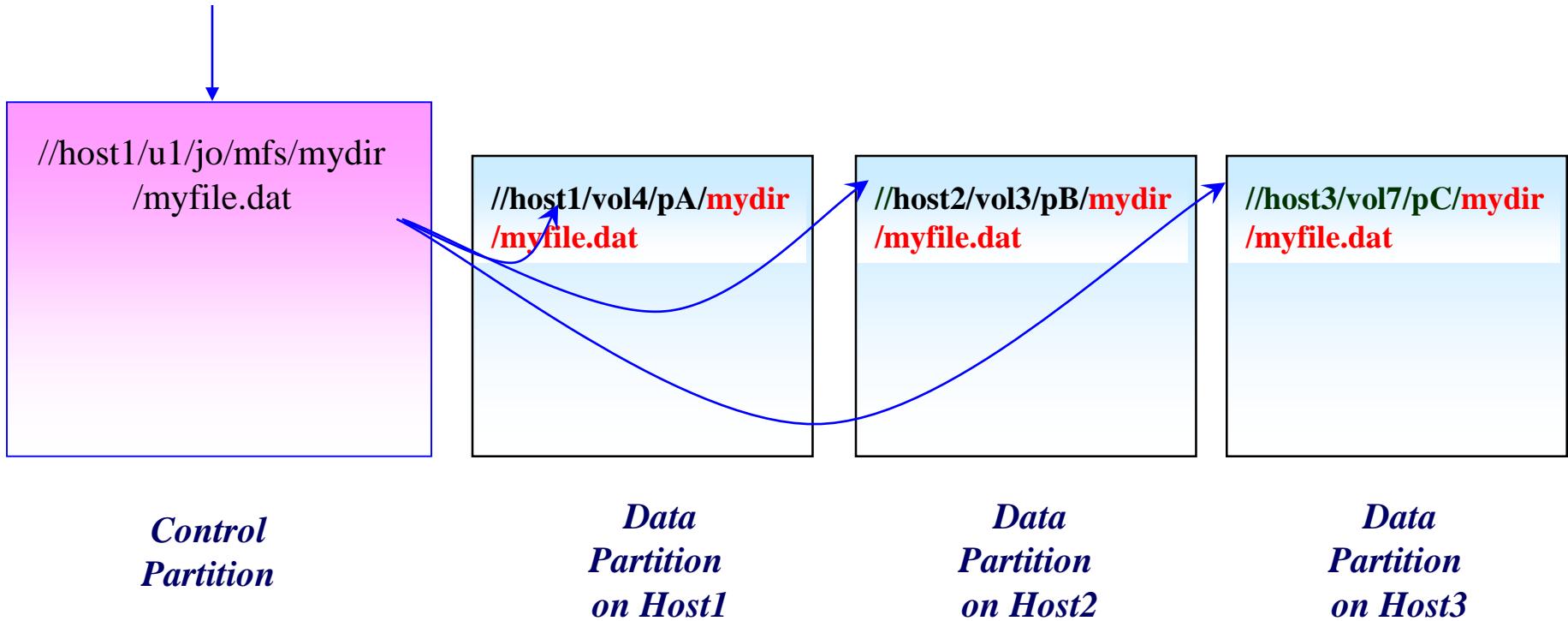
- * A global view of a set of ordinary files called *partitions* usually located on different disks or systems
- * Ab Initio provides shell level utilities called “*m_commands*” for handling multifiles (copy, delete, move etc.)
- * Multifiles reside on Multidirectories
- * Each is represented using URL notation with “*mfile*” as the protocol part:
 - ⇒ mfile://pluto.us.com/usr/ed/mfs1/new.dat

A directory spanning across partitions on different hosts



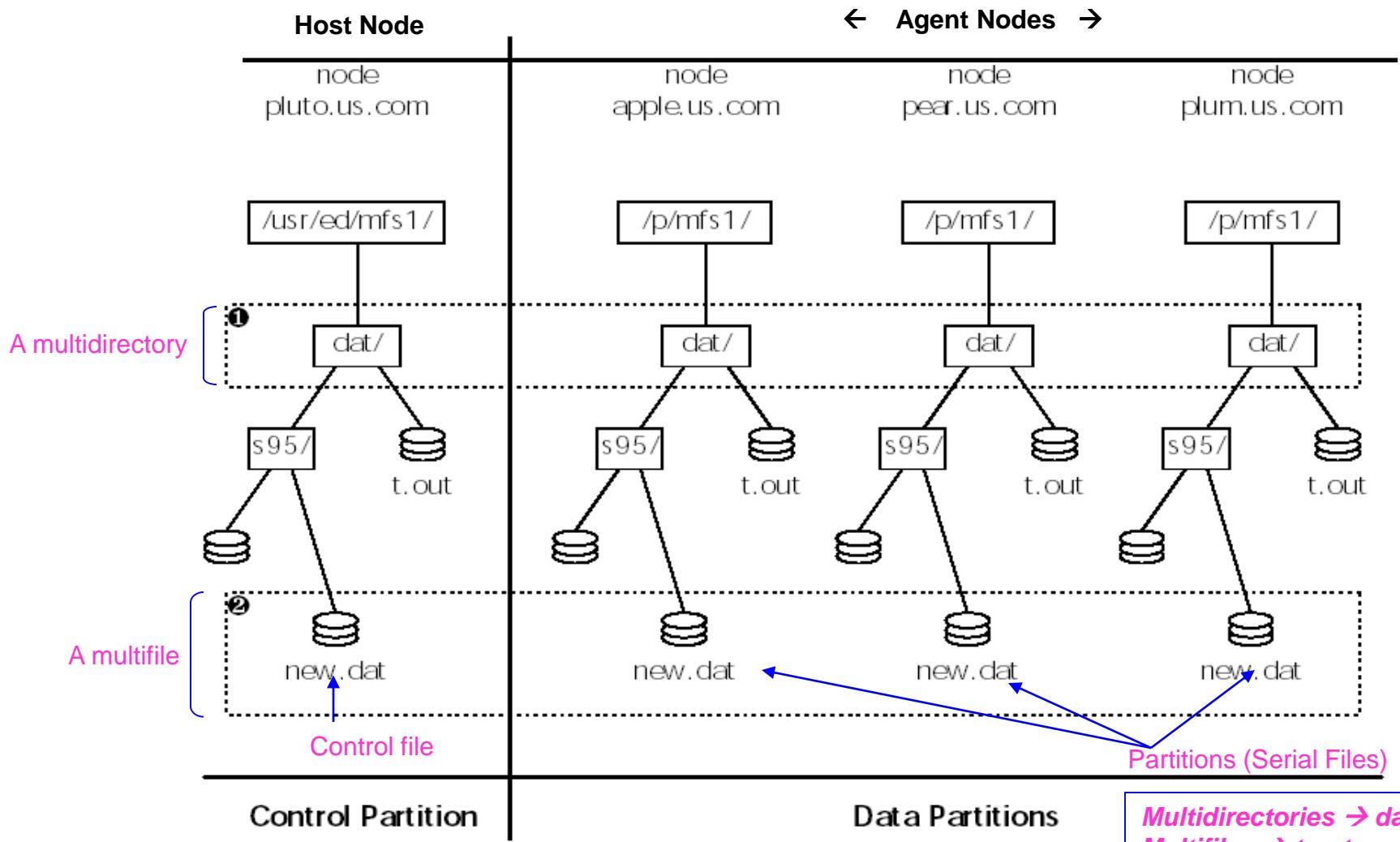
A file spanning across partitions on different hosts

`mfile://host1/u/jo/mfs/mydir/myfile.dat`



A sample multifile system

Ab Initio



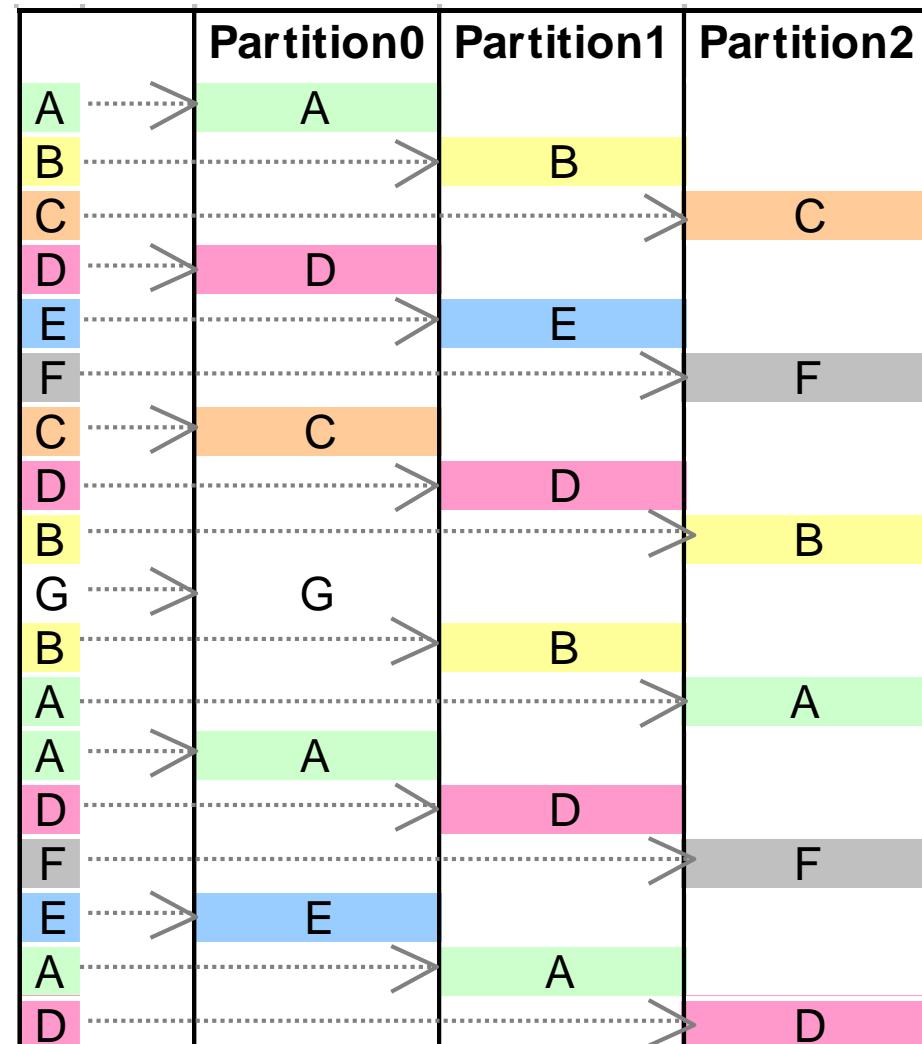
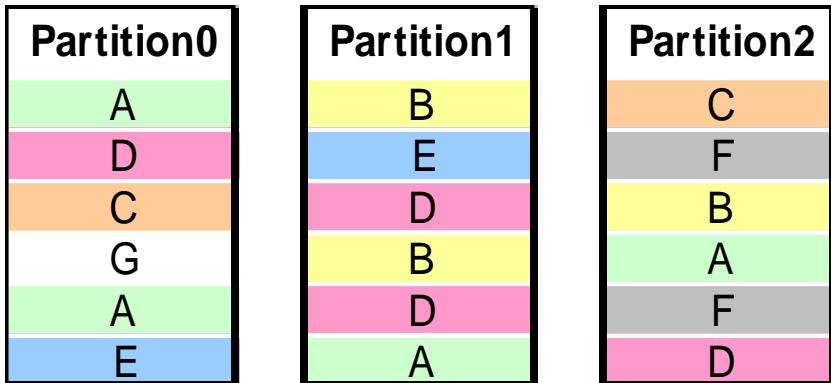
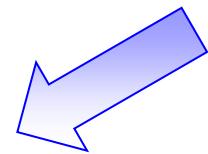
- ★ Data parallelism scales with data and requires data partitioning
- ★ Data can be partitioned using different partitioning methods.
- ★ The actual way of working in a parallel runtime environment is transparent to the application developer.
- ★ It can be decided at runtime whether to work in serial or in parallel, as well as to determine the degree of parallelism

Data can be partitioned using

- * Partition by Round-robin
- * Broadcast
- * Partition by Key
- * Partition by Expression
- * Partition by Range
- * Partition by Percentage
- * Partition by Load Balance

Partition by Roundrobin

Writes records to the flow partitions in round-robin way, with block-size records going into one partition before moving on to the next



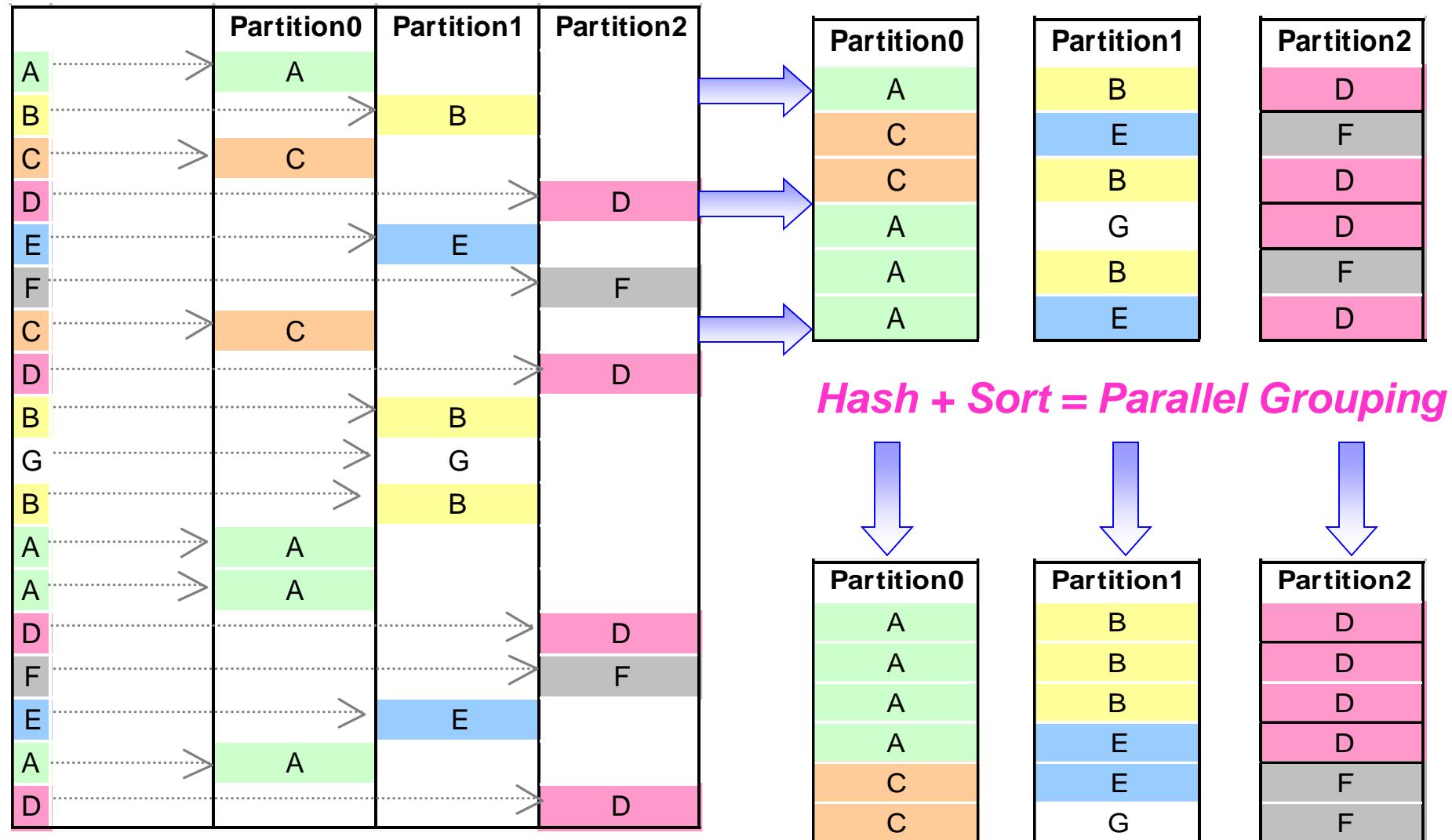
*Key dependent data parallelism requires data partitioning based on key value

*Partition by key

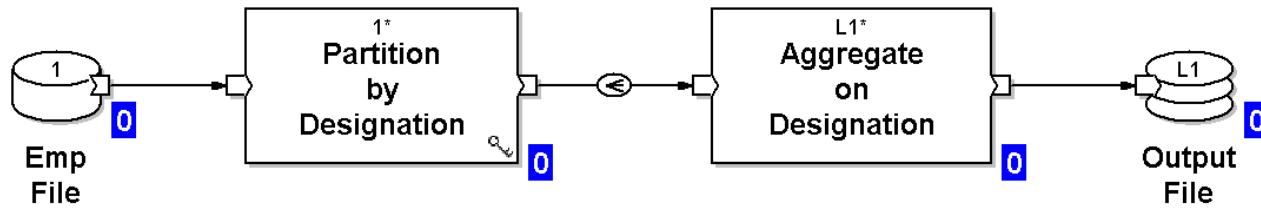
- ✧ Generates a hash code based on the key
- ✧ Hash code determines which partition a record will go into
- ✧ Records with same key value goes into the same partition

Partition by Key is also called Hash Partition

Partition by Key



- * In key-dependent data parallelism, identical key values must be in the same partition
- * Layouts determine the degree of parallelism
- * Fan-out and All-to-All flows with respect to data partitioning



*Partition by Range

- ✧ partitions according to the ranges of key values specified for each partition
- ✧ Splitters or split port
- ✧ Partition by Range + Sort → *Global Ordering*

*Partition by Percentage

- ✧ distributes a specified percentage of the total number of input data records to each output flow
- ✧ Pct port

* Partition by Expression

- ✧ partitions according to a specified hash function or DML expression

* Partition by Load Balance

- ✧ distributes data records to its output flow partitions, writing more records to the flow partitions that consume records faster

* Broadcast

- ✧ Acts like a partitioning component when the layout changes

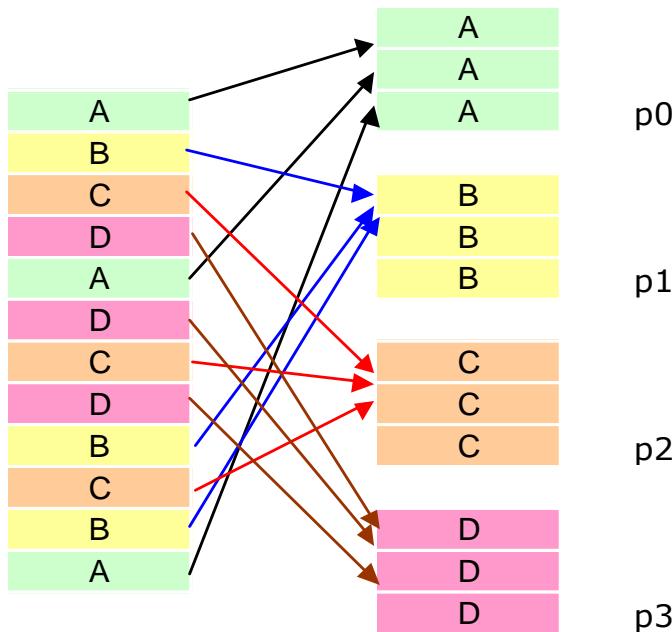
Summary of Partitioning Methods

Ab Initio

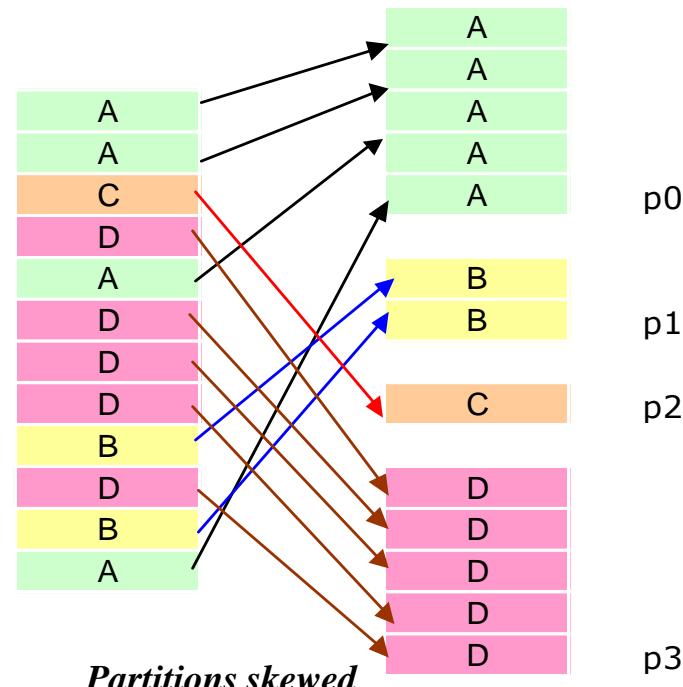
Method	Key-Based	Balancing	Uses
Roundrobin	No	Good	Record-independent parallelism
Hash	Yes	Good	Key-dependent parallelism
Function	Yes	Depends on data and function	Application specific
Range	Yes	Depends on splitters	Key-dependent parallelism, Global Ordering
Load-level	No	Depends on load	Record-independent parallelism

Partitioning Performance : SKEW

Ab Initio



Partitions evenly balanced



Partitions skewed

* Skew determines the distribution of data on different partitions.

* Skew for a data storage partition is defined as: $(N - \text{Average})/\text{Max}$

where N is the number of bytes in that partition

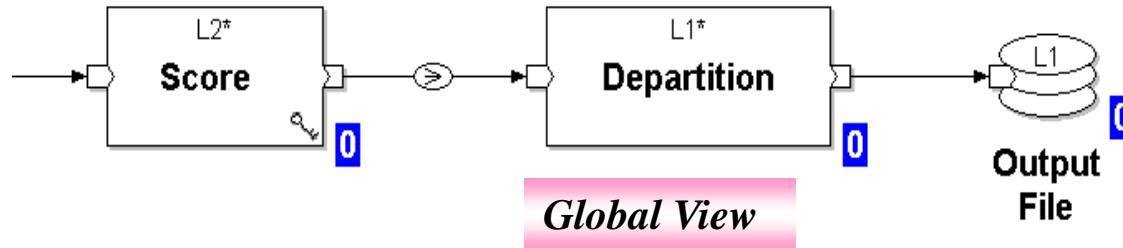
AVERAGE is the total number of bytes in all the partitions divided by number of partitions

MAX is the number of bytes in the partition with the most bytes.

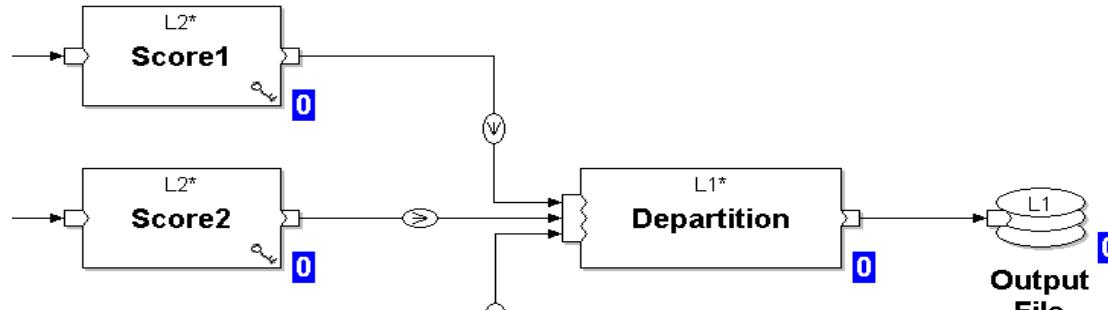
* Common Source of Skew – large key groups

Data can be de-partitioned using

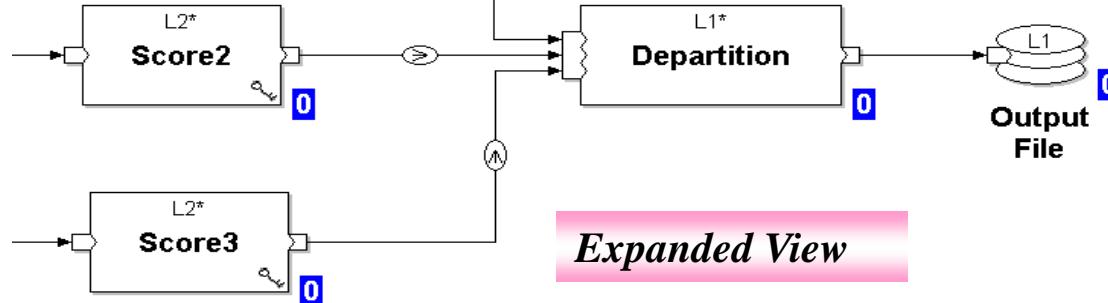
*Gather



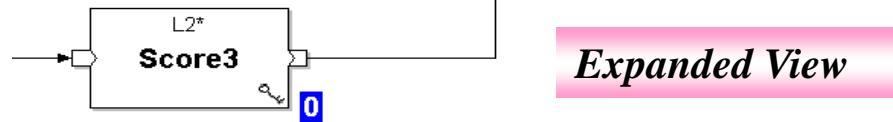
*Concatenate



*Merge



*Interleave



*Gather

- ✧ Reads data records from the flows connected to the input port
- ✧ Combines the records arbitrarily and writes to the output

*Concatenate

- ✧ Concatenate appends multiple flow partitions of data records one after another

*Merge

- ✧ Combines data records from multiple flow partitions that have been sorted on a key
- ✧ Maintains the sort order

*Summary of Departitioning Methods

Method	Key-Based	Ordering	Uses
Concatenate	No	Global	Creating serial flows from partitioned data
Interleave	No	Inverse of Round Robin partition	Creating serial flows from partitioned data
Merge	Yes	Sorted	Creating ordered serial flows
Gather	No	Arbitrary	Unordered departitioning

Ab Initio >

DAY 4

*Serial or Multifiles

*Held in main memory

*Searching and Retrieval is key-based and faster as compared to files stored on disks

*associates key values with corresponding data values to index records and retrieve them

*Lookup parameters

- ✧ Key

- ✧ Record Format

*Storage Methods

- ✧ **Serial lookup : lookup()**
 - ⇒ whole file replicated to each partition
- ✧ **Parallel lookup : lookup_local()**
 - ⇒ file partitions held separately

*Lookup Functions

Name	Arguments	Purpose
lookup()	File Label and Expression.	Returns a data record from a Lookup File which matches with the values of the expression argument
lookup_count()	- do -	Returns the number of matching data records in a Lookup File.
lookup_next()	File Label	Returns successive data records from a Lookup File.
lookup_local	File Label and Expression.	Returns a data record from a partition of a Lookup File.
lookup_count_local()	- do -	Same as lookup_count but for a single partition
lookup_next_local()	File Label	Same as lookup_count but for a single partition

NOTE: Data needs to be partitioned on same key before using lookup local functions

In Parallel Environment unique keys are generated using the functions

- *next_in_sequence()

- *this_partition()

- *number_of_partitions()

Key = number_of_partitions() * next_in_sequence() + this_partition() +
last_generated_key

A Deadlock happens when

- ✧ when the graph stops progressing because of mutual dependency of data among components
- ✧ Identified when record count in does NOT change over a period of time

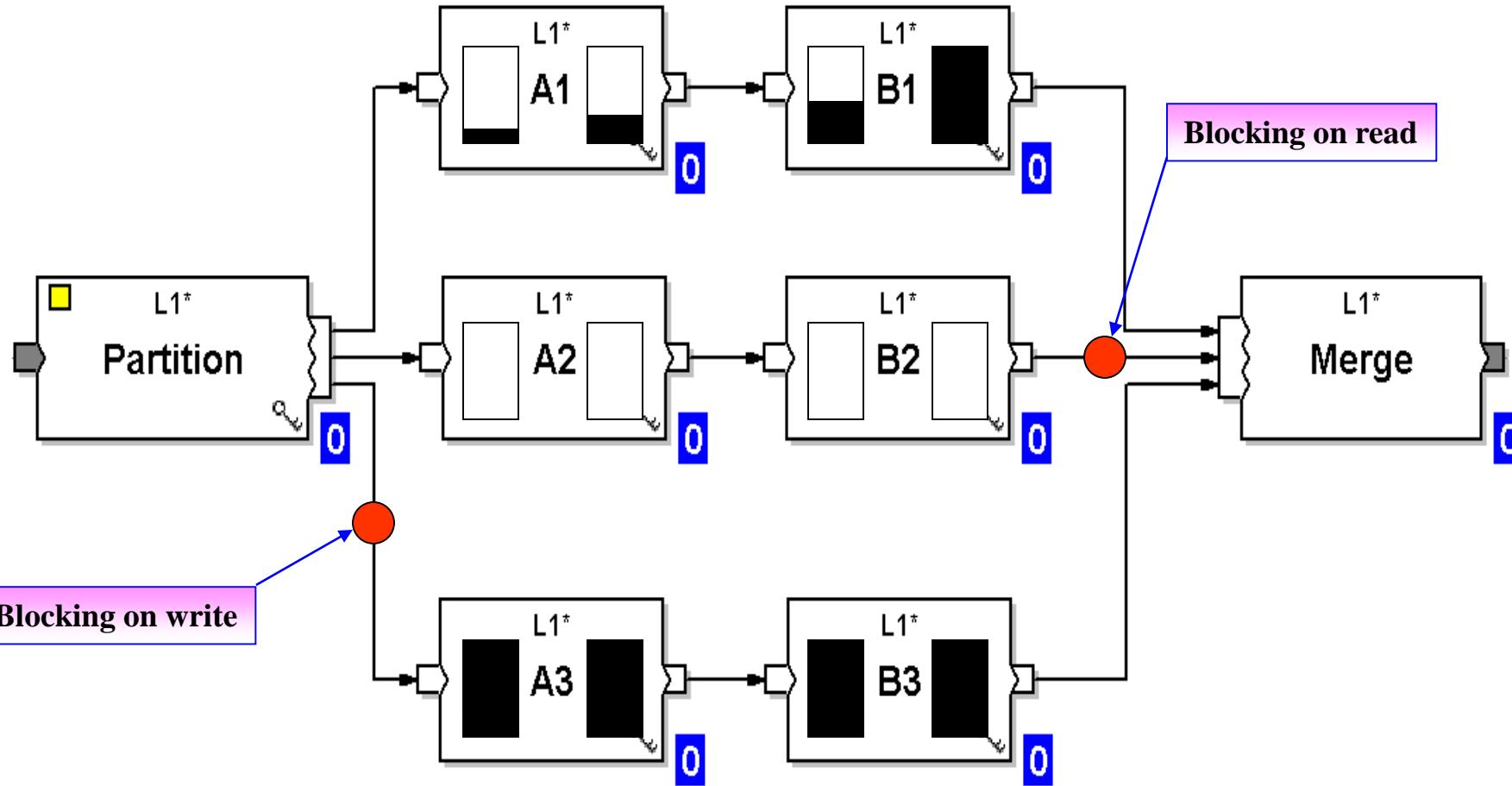
Deadlock can be avoided using

- ✧ Phasing
- ✧ Checkpointing
- ✧ Flow Buffering

*Components likely to cause deadlock

- | | | |
|---------|---------------|---------------------|
| ➤ Join | ➤ Concatenate | ➤ Compare records |
| ➤ Merge | ➤ Interleave | ➤ Compare Checksums |

NOTE : Too many all-to-all flows likely to cause deadlocks



Ab Initio >

Progressing to the Next Level:

Performance

*Data Volume : Files

- ✧ **Source/Target files**
 - ⇒ DB Tables, Flat files
- ✧ **Temporary files**
 - ⇒ Phases
 - ⇒ Checkpoints
 - ⇒ Sort, Merge etc.
 - ⇒ Buffered Flows

*Data Volume : Skew

- ✧ reflects data load balance among partitions

*Data Volume: I/O

- ✧ reading from/writing to files on disk
 - ⇒ datasets
 - ⇒ phases
 - ⇒ checkpoints
- ✧ loading/unloading tables from database
- ✧ "spilling" to disk
 - ⇒ In-memory join or rollup
 - ⇒ sort

*Data Volume: .WORK

- ✧ Temp files in parallel layouts are written to .WORK subdirectories of the multifile system associated with the layout
- ✧ These files are cleaned up on successful job completion or after rolling back the job with m_rollback -d.
- ✧ **WARNING:** .WORK contains hidden files and should never be manually removed and recreated.

*CPU: Number of processes

- ✧ Processes are faster as long as resources are available
- ✧ Overdriving machine
 - ⇒ Resource intensive applications
 - ⇒ Symptoms
 - Increased processes
 - Increased system calls
 - Increased paging of memory
 - Strange error messages

*RULE OF THUMB: GO parallel as soon as possible

NOTE: In Data parallelism # processes per component = # partitions

*Memory: Consumers

✧ Lookup Tables

- ⇒ **Serial lookup: lookup()**
 - whole table replicated into each partition
- ⇒ **Parallel lookup: lookup_local()**
 - table partitions held separately

✧ In-memory components

- ⇒ **Rollup, scan, normalize**
 - (temporary data) x (number of key groups)
- ⇒ **Join**
 - non-driving input(s) loaded into memory

*Memory: max_core

- ✧ Memory Allocated by component to hold data
- ✧ Exceeding max-core
 - ⇒ Disk “spilling” if memory is used up writing to disks
 - ⇒ Graph aborts if memory not available
 - ⇒ Performance bottleneck: slower execution
- ✧ Issues
 - ⇒ Changing data volume over time
- ✧ Rule of thumb : available max-core = (total memory)/(2* # partitions)
 - ⇒ shared among components that allocate max-core in one phase

***Rule of Thumb: GO parallel as soon as possible**

- ❖ Processing data serially causes I/O bottleneck

***Rule of Thumb: Minimize Serial reads**

- ❖ Reading multiple serial reads simultaneously is faster than reading them separately

***Rule of Thumb: Minimize Database access**

- ❖ Unload DB tables once (if possible) – SQL Loader is slow
- ❖ If multiple graphs require same database table, unload to a file and replicate.

*Purpose: Controlling # simultaneous processes

*Performance Enhancements:

- ✧ CPU Usage: Optimum resource utilization - limit the number of active components and thus the number of processes.
- ✧ Memory usage:
 - ⇒ Allocate max-core across components in phase
 - ⇒ Place components which need large max-core in separate phases
 - ⇒ - trade-off: writing to disk at phase boundary
 - ⇒ Sort: Set max-core to 100 MB
 - ⇒ Safety Cushion: DO NOT over allocate memory in a phase

*Performance Enhancements

- ✧ **Communication bottleneck : ALL-to-ALL flows**
 - ⇒ **N-way to N-way: Uses N^2 network resources**
 - ⇒ **Limit the number of All-to-All flows by using phases**
 - ⇒ **Safety cushion: <=4 per phase**
 - ⇒ **Alternative: Use Two-Stage-Communication**
 - **Uses $2Nx\sqrt{N}$ channels of communication**
 - **applicable if depth is $>= 30$**

*Rule of thumb: Placement of phases

- ✧ DO NOT put phases after Replicates, across all-to-all flows, after temp files, after sorts
- ✧ Data on all flows crossing the current phase boundary is written to disk, so place phases to minimize writing of data to disks

*Purpose:

- ✧ Provide same functionality as phase
- ✧ Additional: Provide restart capability

*How does it work ?

- ✧ At job start, output datasets are copied to temporary files (in .WORK dirs)
- ✧ At checkpoint completion, intermediate datasets and job state are stored in temporary files
- ✧ Recovery information is stored in host and vnode directories of /var/abinitio

*Rule of thumb : Placement of Checkpoint

- ✧ Same as phase

*Rule of thumb : Minimize the amount of data to be held on temporary storage

*Rule of thumb : Recovery files

- ✧ Should be explicitly deleted/rolled back when a checkpointed graph is aborted
 - ⇒ `m_rollback -d graphname.rec`
 - ⇒ **WARNING:** rolling back old .rec files will restore the output files to old state

- ★insert temporary files to check/capture intermediate data
- ★set reject to Never Abort, capture and inspect rejected records
- ★attach Logger component and inspect log messages
- ★advanced: use `write_to_log()` / `validate` to debug dml code
- ★advanced: use `hash_value()` to check partitioning
- ★Use check order component to check sorting
- ★Using `m_dump` command to view data

Performance: Debugging Log file

Ab Initio

A sample log file ..

```
more produce.log

Mar 15 17:46:47    Phase 0 started <0 seconds>
                    CPU Time Status Skew Vertex
                      0.100 [ 1: ]  0% Redefine_Format
                      0.040 [ 1: ]  0% Reformat
                    Data Bytes      Records      Status      Skew Flow Vertex      Port
^L

Mar 15 17:46:48    Phase 0 running <1 second>
                    CPU Time Status Skew Vertex
                      0.210 [ : 1]  0% Redefine_Format
                      0.170 [ 1: ]  0% Reformat
                    Data Bytes      Records      Status      Skew Flow Vertex      Port
                     10,569        136 [ : : : 1]  0% Flow_1 Reformat in
                     10,002        136 [ : : : 1]  0% Flow_2 Reformat out0
                     10,002        136 [ : : : 1]  0% Flow_2 Redefine_Format in
                     10,002        136 [ : : : 1]  0% Flow_3 Redefine_Format out
^L

Mar 15 17:46:48    Phase 0 ended <2 seconds>
                    CPU Time Status Skew Vertex
                      0.210 [ : 1]  0% Redefine_Format
                      0.170 [ : 1]  0% Reformat
                    Data Bytes      Records      Status      Skew Flow Vertex      Port
                     10,569        136 [ : : : 1]  0% Flow_1 Reformat in
                     10,002        136 [ : : : 1]  0% Flow_2 Reformat out0
                     10,002        136 [ : : : 1]  0% Flow_2 Redefine_Format in
                     10,002        136 [ : : : 1]  0% Flow_3 Redefine_Format out
^L
[produce.log]<EOF>
```

*Reading the Log : CPU

- ✧ CPU time: total processing for component
- ✧ Status: [Running : Finished]
- ✧ Skew: among CPU times of each partition
- ✧ Vertex: component

*Reading the Log : DATA

- ✧ Data bytes: # processed
- ✧ Records: # processed
- ✧ Status: [unopened : opened : closed]
- ✧ Skew: among data bytes in partitions

*Reading the Log : DATA

- ✧ Flow: link between components
 - ⇒ data tracking info is displayed on flows in GDE
- ✧ Vertex: component
- ✧ Port: of component

*Interpreting the log

- ✧ Compute data bytes/sec through component, in each partition
- ✧ Look for serialization: effective CPU = $(\text{cpu time}) / (\text{elapsed time})$
- ✧ compare open vs. closed partitions: serialized when some partitions remain open long after others have closed → data skew
- ✧ Deadlock: no change in record counts over couple of intervals

*Avoid Sorts

*Use Lookups

*Use In-memory Join/Rollup

*Assign Driving Port of Join correctly

*Allocate memory correctly

*Phasing

THANK YOU