



Release 2.0.0 (Production)

**NOTE: PLEASE DO NOT
DISTRIBUTE OR QUOTE**

I. Change Log

New Chapters

- Operators (Ch 12) is now flushed out

Modified Chapters

- Declaration (Ch 3)
 - Labeled input/output types of TheaterLoc sample plan
 - Metadata example added with current Theseus operator API
- Invocation (Ch 4)
 - Examples modified to use `xmlcli` as our XML command line client
- Termination (Ch 7)
 - The most common way that a plan terminates was glossed over in earlier versions. There is now a more complete description of the detection of exit flows and the voting process for terminating a transaction at both the operator and at the plan levels.
- Implementing the Executor (Ch 11)
 - In the basic threaded model, consumer queues do not poll, they `wait()`.
- Implementation Notes (Ch 14-16)
 - has been collapsed into one chapter (14)
- Component Development
 - has been deleted

Responses To Feedback

- *No feedback to respond to*

Still On The Stack

- Plans in the filesystem/database
 - *waiting on integration with the registry*

II. Preface

Purposes of This Document

This document describes the design and implementation of Theseus, an information agent plan execution system. **Its main purpose is to represent an agreement between the designers and implementers of Theseus about various architectural decisions that have been made.** The hope is that the end product can be compared to what was promised (this document) so that any missing or errant functionality can be remedied.

A second purpose of this document is to force the designers to specify exactly what is required. Design meetings often leave out the details and get lost in bouts of hand-waving specifications of functionality. This document forces the designers to elicit the details of their specifications.

This document is also intended to be useful for future design and development. Specifically, future designs will be able to easily identify what the vision was for the previous version was, as well as what was actually implemented. When solidifying the vision for a future version, it is often helpful to reconsider design decisions made in the past.

Intended Audience

It is assumed that readers of this document have a computer science background – particularly intelligent agents, operating systems, artificial intelligence, databases, and Internet technologies. Although this document does contain some introductory material, it is a “design and implementation” document and not meant for a general audience.

1. INTRODUCTION.....	8
1.1. <i>What Is Theseus?</i>	8
1.2. <i>Input, Output, and What Happens In Between</i>	9
1.2.1. <i>Input</i>	9
1.2.1.1. <i>Information Agent Plans</i>	9
1.2.1.2. <i>Input Data</i>	10
1.2.2. <i>Output</i>	10
1.2.3. <i>What Happens In Between</i>	10
2. TECHNICAL OVERVIEW.....	11
2.1. <i>Plan Language and Dataflow Architecture</i>	11
2.2. <i>Streaming</i>	12
2.3. <i>Transactions</i>	12
2.4. <i>Looping, Iteration, and Recursion</i>	13
3. DECLARATION.....	15
3.1. <i>Plans</i>	15
3.2. <i>Datatype Classifications</i>	15
3.3. <i>Datatypes</i>	16
3.4. <i>Plan Language</i>	17
3.4.1. <i>Variable Input/Output Arguments</i>	20
3.5. <i>Naming and Scoping</i>	21
3.6. <i>Declaring and Defining Operators</i>	22
3.6.1. <i>Operator Metadata</i>	22
3.6.2. <i>Operator Code</i>	24
4. INVOCATION.....	25
4.1. <i>Methodologies</i>	25
4.1.1. <i>Invocation via Network (Server Mode)</i>	25
4.1.2. <i>Invocation via Command Line (Interactive Mode)</i>	26
4.2. <i>Preparing For Invocation</i>	26
4.2.1. <i>Plan Lifecycle</i>	26
4.3. <i>Plan Optimization</i>	27
4.4. <i>Plan Compilation</i>	27
4.5. <i>Plan Instantiation</i>	28
4.5.1. <i>Input and Output Adapters</i>	28
4.5.2. <i>Input and Output Formats</i>	28
5. THE SYSTEM.....	31
5.1. <i>Components, Tools, and Services</i>	32
5.1.1. <i>The Registry</i>	32
5.1.2. <i>The Alarm Service</i>	33
5.1.3. <i>Command Line Client</i>	33
5.1.4. <i>Web Server (HTTP Server)</i>	34
5.1.5. <i>Plan Compiler</i>	34
5.1.6. <i>Plan Loader and Unloader</i>	34
5.1.7. <i>External Database</i>	34
5.1.8. <i>Plan Service</i>	34
5.1.9. <i>Plan Scheduler</i>	34
5.2. <i>An Important Note About the Components</i>	35
5.3. <i>Examples of Using The System Components</i>	35
5.3.1. <i>Interactive Execution, Assuming Nothing</i>	35
5.3.2. <i>Interactive Execution for Deployed Plan</i>	35
5.3.3. <i>Serving a Deployed Plan</i>	36
5.3.4. <i>Invoking a Served Plan Remotely</i>	36
6. EXECUTION.....	38
6.1. <i>Summarizing Execution</i>	39
6.2. <i>Streaming</i>	39

6.3. Transactions	40
6.3.1. Transaction Management	40
6.3.2. Transaction Lifetimes	40
6.4. Operator Firing Rule	41
6.5. Operator Processing	42
6.6. State Management	43
6.7. Subplans	44
6.7.1. Transactions and Subplans	45
6.7.2. Tail-Recursive Optimization	45
7. TERMINATION	46
7.1. Plan Termination	46
7.2. Transaction Termination	46
7.2.1. Explicit Termination	46
7.2.2. Implicit Termination	47
7.3. Detecting Transaction Termination at the Plan Level	47
8. ERROR HANDLING	49
8.1. What Are Errors?	49
8.2. Default Error Handling	49
8.3. Plan-Specific Error Handling	51
9. INSTRUMENTATION AND DEBUGGING	52
9.1. Events	52
9.2. Callbacks	53
9.3. Event Queues & Listeners	53
9.4. Generating Events and Callbacks	53
10. OPTIMIZATIONS	54
11. IMPLEMENTING THE EXECUTOR	56
11.1. Aspirations	56
11.2. Theoretical Parallelism vs. Real Parallelism	56
11.3. Executor Implementations	56
11.3.1. The Basic Threaded Implementation	57
11.3.2. The Manager/Workers Implementation	58
11.3.2.1. Resource Allocation Issues (Transaction Concurrency vs. Parallelism)	60
11.3.2.2. Operator I/O Details	61
Combing Event-Driven and Polling Approaches	61
Ring-Based Queue Processing	61
The Queue-Offset Method (<i>Alternative Approach</i>)	63
11.3.2.3. Dedicated Worker Pools vs. Global Worker Pool	65
11.3.2.4. Work Stealing	65
12. PLAN OPERATORS	67
• <i>Stream Operators</i>	67
• <i>Relational Operators</i>	70
• <i>Database Operators</i>	72
• <i>Utility Operators</i>	73
• <i>Misc Operators</i>	75
13. DEVELOPING OPERATORS	76
13.1. Writing Queue Processing Functions	76
14. NOTES	78
14.1. Streaming	78
14.1.1. Shallow Copying of Tuples	78
14.1.2. Relation Identifiers	78
14.2. Notes on Subplans	79
14.2.1. Splicing Plans Together	79
14.2.2. Scaling Worker Threads	79
14.3. Notes on State Management	79

15. EXAMPLES	80
15.1. <i>Operator Implementation Example</i>	80
15.2. <i>Plan Examples</i>	82
15.2.1. HomeSeekers	82
15.2.2. Yahoo Spidering	83
15.2.3. TheaterLoc	84

INTRODUCTION

This part of the document introduces Theseus and the concepts of dataflow-based plan execution for information agents. It also focuses briefly on describing some of the efficient technologies used in the design and how they were motivated.

1. Introduction

1.1. What Is Theseus?

Theseus is an efficient execution system for information agents. More specifically, Theseus empowers users by (a) simplifying the process of *specifying* agents that can manage and monitor dynamic data sources, such as web sites, and (b) provides a flexible, optimized platform for agent execution.

To understand the value of managing and monitoring dynamic sources, consider the difficulty in using the Web to locate a new house that meets a set of criteria (better known as the HomeSeekers example). To accomplish this task, a user is required to manually “keep track of” a set of online sources over a period of (usually) weeks. Specifically, the user must:

- learn how to use the different user interfaces at each web source
- remember which houses he has already looked at (it is a waste of time to look at them again and again)
- prioritize which houses most closely meet his criteria
- filter out duplicates that exist across multiple web sites
- manually extract matching houses from those sources that do not allow him to specify his criteria (i.e., suppose he is only looking for houses in Santa Monica that are no more than 10 years old)

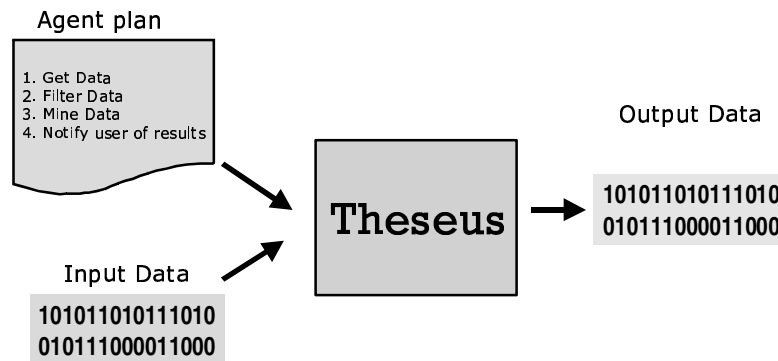
Again, all of this must be done repetitively, over the course of (usually) weeks. The worst part about this process is that the user is the real bottleneck: most people do not have lots of time to monitor multiple web sources, much less distinguish duplicate houses between them, etc. This type of automated information management is exactly what Theseus aims to address.

To solve dynamic information management problems, such as the HomeSeekers example, with Theseus, a user would specify a simple *plan* to accomplish the task. For HomeSeekers, the plan would describe the retrieval and management of houses from various web sites. The “management” part of the plan refers to the determination of houses that meet the user criteria, as well as the ability to distinguish new houses from ones already seen¹.

¹ Of course, while Theseus makes it easy to specify this type of plan, most users will want a GUI to help construct the plan automatically. Nevertheless, having a simple plan language allows GUIs to easily generate plan code as well as making it easy for developers to hand-write plans.

1.2. Input, Output, and What Happens In Between

In general, Theseus requires information agent *plans* and a set of input data and returns data back to the user as output. The figure below illustrates this:



1.2.1. Input

1.2.1.1. Information Agent Plans

Plans specify the retrieval, integration, management and reporting of data back to users. They are composed of *operators* that each has a role in processing the data – either in terms of its retrieval, its output, or some intermediate-processing task. Just as in standard AI planning systems, operators are triggered by the fulfillment of pre-conditions. In the case of Theseus plans, these pre-conditions are data: either that which is input into the plan or that which is produced by some other plan operator. Thus, operators in Theseus are said to both *consume* input data and potentially *produce* output data.

Depending on the need, plans can be either run *on-demand* or scheduled for *continuous execution*. Running plans on demand allows users to integrate and analyze data immediately. For example, consider a plan that locates the restaurants and theaters, along with movie showtimes, in a user-specified city and plots them on a map (better known as the TheaterLoc example). This type of plan is powerful because it integrates multiple data sources (restaurant source, theater source, and map source) together and returns back a simple, yet useful result. However, such plans are usually executed on demand. Users rarely wish to monitor web sites for restaurants and theaters – more commonly, they want to execute this query every Friday night to see where they should go for dinner and a movie. In contrast, continuously executing plans enable true asynchronous monitoring: the user does not regularly invoke them. These plans run periodically until requested to stop. The HomeSeekers example (described earlier) is one such example.

Plans can also be set up to run either for a single user or so that they can be shared by multiple users. For example, with TheaterLoc, users could each run their own TheaterLoc plans through their own Theseus executors to obtain results. This would be the single-user case. However, it would require that each user have access to (or write) such a plan. Furthermore, each user would be required to have a copy of the Theseus system. In such situations, it is more efficient to deploy the TheaterLoc plan so that it can be shared by multiple users – since the only thing that changes for each user is the city he/she wishes to query, it is simpler to just allow users to interface with a continuously available plan at a well-known network address (i.e., a URL). This

allows all types of clients to interface with Theseus without having to get a copy of all of the resources necessary for execution.

1.2.1.2. Input Data

The initial set of input data and the existence of a valid plan are what trigger the execution of Theseus. As will be described in more detail in Chapter 2, Theseus executes plans under a *dataflow* model of computing. The initial set of input data enables some initial set of operators to execute. These, in turn, trigger other operators to execute, and so on until no operators remain executing. Thus, the initial set of input data is very important to execution.

Some plans may not have an initial set of input data – they may just execute a task. These types of plans are usually scheduled for periodic execution.

1.2.2. Output

The result of Theseus plan execution is usually data. For example, the HomeSeekers plan periodically returns a list of new houses that meet the users' criteria. Likewise, the TheaterLoc plan returns a map of movies and restaurants in a given city.

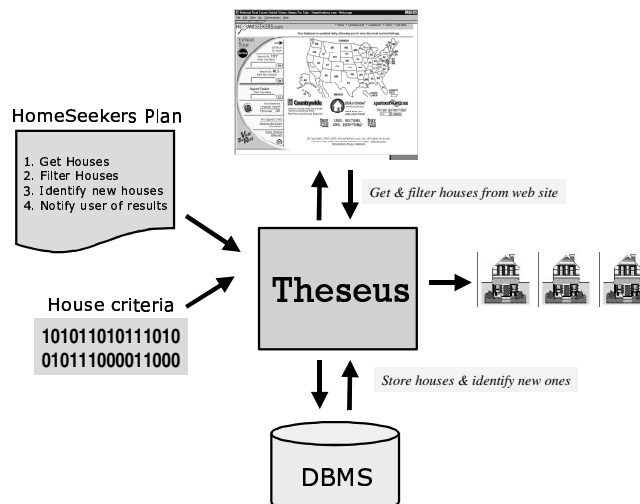
Output from Theseus can be encoded in ASCII or a variety of other formats (HTML, XML), depending on how the plan was invoked. Also, since plans can be running continuously, monitoring web sites, Theseus allows data to reach the user using a variety of asynchronous mechanisms (e-mail, FAX, pager).

While most plans output data when they are executed, this is not always the case. Continuous information management often means tending to housekeeping or encountering no new results from a particular search. With HomeSeekers, for example, a plan that is scheduled for periodic execution may find new houses 10% of the time it executes (perhaps the user has very restrictive criteria or plan is executed frequently).

1.2.3. What Happens In Between

The Theseus agent execution platform acts like a black box, taking the input plan and returning the data it gathers as output. However, as mentioned earlier, these plans may be written so information is gathered from remote sources and “intelligently” managed, often out of sheer necessity.

For example, in the HomeSeekers plan, Theseus spends some of its execution time retrieving data from the HomeSeekers web site, as well as interacting with a local database management system (DBMS) to keep track of the houses it has sent to the user. This is shown in the figure below. In this way, users will not receive duplicate notifications about the same house.



2. Technical Overview

This part of the document assumes some familiarity with the details of Theseus V1, described further in [Barish, DiPasquo, Knoblock, and Minton 1999].

2.1. Plan Language and Dataflow Architecture

As mentioned earlier, two key goals of Theseus are to (a) optimize the execution of data integration plans and (b) extend the notion of on-demand information integration to the concept of *autonomous information management*. Traditional data integration technologies have been effective, but can sometimes appear slow because the network (and disk) latency between the heterogeneous data sources dominates the time of execution. Furthermore, although data integration has enabled users to collapse widely distributed data in a unified view, little research has been done on designing agents that do anything beyond this. Specifically, no research has been done on optimizing the process of autonomous data integration, management, or otherwise monitoring of the sources that produce this data.

The Theseus plan language and system architecture has been designed specifically with the above goals in mind. The language allows complex information management plans to be easily specified. Theseus operators support standard relational operations on data (i.e., Select and Join), management of acquired data and metadata in local external databases, continual periodic querying, and flexibility in the communication of those results (i.e., e-mail). In short, the Theseus plan language extends the typical set of operators used in data integration systems to help support the goals of autonomous execution and information management.

The Theseus architecture is based on a dataflow-style of execution, enabling information agent plans to achieve a high degree of parallelism. Theseus plan operators execute when they are enabled, allowing partially ordered plans to realize parallelism as the preconditions to those operators (data) become available. Furthermore, the architecture supports the asynchronous pipelining of data, so that groups of tuples can be consumed and produced inexpensively by operators - reducing the cost of operator synchronization and allowing sequential processing pipelines to be more efficiently implemented. This type of parallelism is especially important in data integration environments, where variable network latencies invalidate assumptions about plan ordering and encourage architectures that (a) react to data as soon as it becomes available and (b) support bursty retrievals of information.

Theseus executes plans that are specified as dataflow graphs, specifically a network of operators that pass data between each other. Operators can be thought of as finite state machines that, when enabled, perform a specific type of information management action. In Theseus V1, all of the input data required by an operator needed to arrive at the operator before it could fire. In Theseus V2, to support streaming, we have adopted a variable firing rule policy, where operator firing can occur in the presence of partial inputs. This is described further in the “Execution” part of this paper. Operator inputs in Theseus are similar to planning pre/post-conditions, except that they can also carry data, thus allowing information to be easily routed through the plan. Declaring plans with operators that produce and consumed named data allows execution to be specified succinctly, only in terms of those control and data dependencies necessary to ensure correct execution.

Theseus operators include those useful for data processing, remote information retrieval, local storage (i.e., in a local relational database), and those for flexible communication of plan results (i.e., via e-mail). Plans leverage from these operators and built-in support for loops and conditionals so that powerful, practical information management plans can be specified. In

Theseus V1, cycles in the dataflow graph were supported. In Theseus V2, cycles are not supported – all plans are directed acyclic graphs (DAGs). Looping paradigms are instead expressed through subplans (both recursive and non-recursive).

The Theseus execution engine is essentially designed to function like a hybrid dataflow machine [Papadapolous and Traub 1991] [Ianucci 1988]. The availability of data determines when various operators execute. Thus, as is the case with most dataflow systems, parallelism and synchronization are realized automatically. Operators are implemented as threads, so Theseus can theoretically achieve as much true parallelism as there are CPUs. The execution system also supports *pipelining*, in which producer operators asynchronously propagate data to consumer operator queues. Without the synchronization overhead, opportunities for parallel execution are increased.

Through its language and execution system, Theseus enables agents to perform useful information management tasks, such as periodic execution, query result accumulation, and flexible result communication. Most importantly, through properties of its architecture, Theseus reduces the overall effect of network latencies on data integration, providing increased parallelism and asynchrony during execution so that the overall end-to-end agent execution process is substantially faster.

2.2. Streaming

Theseus primarily is concerned with the retrieval, processing, and management of data in the form of relations. Since relations are made up of one or more tuples, and retrieval of these tuples may be a time-consuming process, it is desirable to be able to process them as they become available. For example, if a Theseus plan involves retrieving a list of all books from Amazon and then filtering out only those that are mystery books, it would be optimal if the filtering occurred on the tuples as they were extracted (rather than waiting for all of them to be extracted). The concept of communicating data (particularly relations) between operators as tuples become available is known as *streaming*. The key advantage of this type of processing is that it exploits parallelism better: since retrieval is a time-consuming I/O-bound operation, the CPU is available to handle filtering while the I/O continues. More generally, a given relation can have its tuples processed in parallel by as many operators in the pipeline as can be applied. This may depend on the size of the relation or it may depend on the data dependencies of the operators in the graph.

Theseus V2 supports the streaming of relations, the non-streaming of other data types, and the declaration of static (non-moving) data.

2.3. Transactions

Plan execution in a dataflow architecture is such that many operators may be executing in parallel, asynchronously processing intermediate results, on the basis of a single invocation. The lack of coordination between these operators is intentional; dataflow architectures are designed to make the scheduling of parallelism automatic and lightweight. In this sense, operators act much as processes in a distributed system, where management of a logical transaction is decentralized.

In Theseus, the invocation of a plan – from the presentation of input until the reception of output – is considered a logical *transaction*. In Version 1, Theseus plans were run as servers – an invocation was thus a request by client that the “plan server” process input. Upon invocation, the input, intermediate results that were generated, and eventual output were associated with a default transaction. Furthermore, there was an implicit assumption that these plan-servers could only be invoked one at a time. This was because there was a no mechanism to distinguish data

between concurrent, but distinct executions, since an in-process invocation already owned the default transaction.

Theseus V2 manages transactions for both concurrent execution and data management.

2.4. Looping, Iteration, and Recursion

Theseus plans often demand cycles, since information integration is usually an iterative, looping process. Typically, the reason has to do with the interleaving of navigation with extraction. For example, when retrieving a list of mystery books from Amazon, the web site usually provides a set of 25 results before including a link to “the next page of results”. Thus, the information gathering process is usually something like:

1. Query the web site
2. Extract the data items presented on the results page
3. Check if there is a “NEXT PAGE” link
4. If so, repeat step 2, else QUIT

Cycles in this kind of plan are referred to as *loops* in Theseus- iteration through a set of data (in the above case, “next page” links).

Some plan loops are *bounded* because it is clear – in advance – how many iterations will be necessary. For example, when querying a web site, we may be presented not only with the initial set of results, but links to all of the results web pages. Other loops are *unbounded*, such as those in which the gathering process involves following a “NEXT PAGE” link, until we reach the last page.

Note the difference in potential parallelism in these two cases. In the bounded-loops case, we can effectively gather results data simultaneously, once we get back the first page of results. Since all links are presented at once, we can imagine pursuing them at once. In contrast, the unbounded-loops case demands parallelism in somewhat of a pipelined manner. We can only gather data from a results page when we gather data from the previous page. Of course, this could be optimized by having two types of retrievals – one that extracts results and one that extracts only the “next page” information – but the resulting execution would still be result in less parallelism demands than in the bounded-loops case.

Finally, complicating loops (both bounded and unbounded) are the dual methods of dealing with the data extracted via these loops. Some plans require that data be *accumulated*, gathered in whole, before doing some sort of processing (such as e-mailing that data or sorting). Other plans simply want to leverage the streaming implicit in the system so that the data can be processed as it becomes available.

Theseus V2 supports both bounded and unbounded loops for both accumulation and non-accumulation scenarios.

DESIGN

The design part of this document attempts to describe the abstract design of Theseus. No assumption is made about any implementation approach or specific technology.

3. Declaration

Theseus process plans written in the Theseus Plan Language. Plans can either be written by users/administrators, or generated automatically from Ariadne or the Theseus GUI.

In the following section, we first describe the purpose of plan. Then, we introduce the Theseus datatypes and their classifications before describing the plan language in greater detail. We also include information on how plans should be named and the namespace of plans in the system. Finally, we describe an abstract Theseus plan operator and the metadata it requires.

3.1. Plans

As mentioned in Chapter 1, plans in Theseus are simply the instructions for how to integrate, process, manage, and communicate data that is retrieved from multiple dynamic data sources. Specifically, a plan consists of (a) a set of input data, (b) a directed acyclic graph (DAG) of plan operators, and (c) a set of output data. Operators can consume data and can also produce data during execution. For example, a Theseus Retrieve operator might require a query as input and return data as output. All data objects – whether used as input, produced intermediately by operator(s), or output – are uniquely named.

There are two additional plan requirements that Theseus enforces in order to ensure that plans are “well formed”. First, while there can be zero or more input/output data objects, but there must be at least one plan operator. Second, all input objects must be consumed and some operator must produce all output objects in the plan.

Plans also function as operators – in this sense, invoking a plan like an operator means calling that plan as a *subplan*. We will describe subplans in more detail later, but it is sufficient for now to say that they can be called directly from a plan by name.

Before specifying the details of the plan language, it is first necessary to describe the type of data objects that Theseus processes.

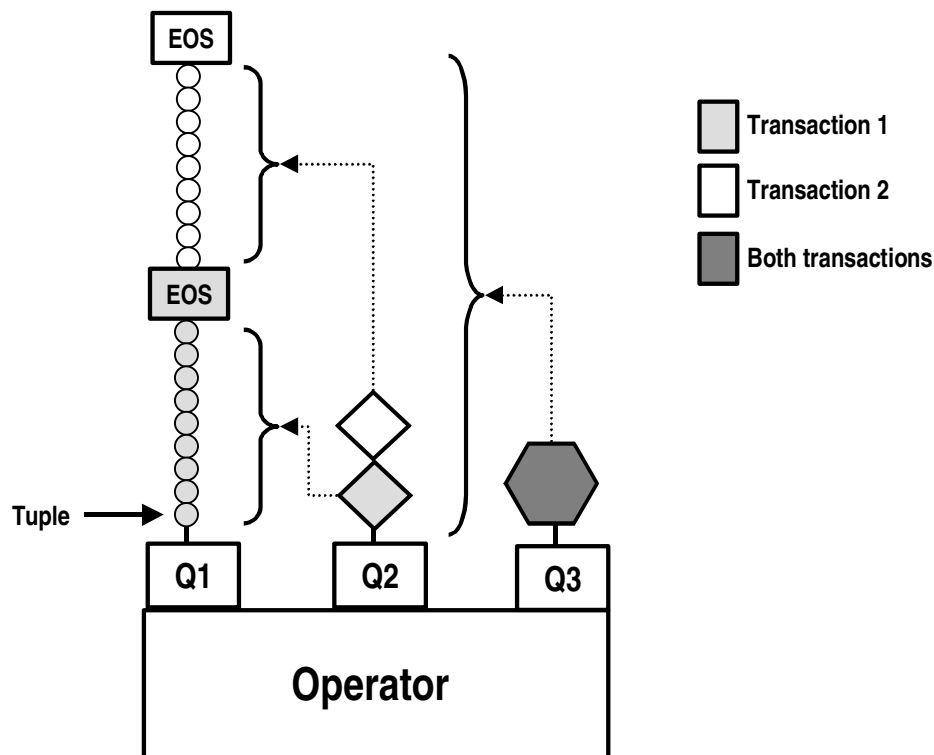
3.2. Datatype Classifications

The following classifications of datatypes are supported in Theseus V2:

- **streaming** objects (relations)
- **per-stream** objects (dynamic, but correspond to a particular stream)
- **static** objects (permanent and correspond to all streams)

The notion of streaming is central to Theseus V2. Often, operators contain a mix of streaming and non-streaming objects. Under these circumstances, per-stream objects can be thought of as valid for the duration of the other streaming inputs – that is, until an EOS is seen on all of those streams. Note that since Theseus plans are DAGs, there is no chance that the same transaction will send multiple EOS at a given operator for a given transaction. Static objects are a special instance of a per-stream object. They can be seen as similar to persistent enablements in Theseus V1 – they continue to trigger the firing rule for the operator and they are applied to all streams.

For example, the figure below shows an operator that has three input queues: streamed (Q1), per-stream (Q2), and static (Q3).



In the figure, notice that the Q3 data is coordinated with all Q1 and Q2 data. Also, the first Q2 object (corresponding to the first transaction) is coordinated with Q1 until the first Q1 EOS is encountered. Then, the current Q2 object is flushed and the next Q2 object (corresponding to the next transaction) is applied to the next stream of Q1 (until the next EOS is seen, and so on). Meanwhile, Q3 data applies to both transactions.

3.3. Datatypes

There are five default datatypes in Theseus V2:

- TupleStream (streamed)
- Relation (per-stream)
- String (per-stream)
- Number (per-stream)
- Enablement (per-stream)

A TupleStream is the only default streamed datatype in the system. It is simply a stream of tuples that logically belong to the same relation. Thus, they each have the same attribute list (names and types). As with any stream, communication of a Tuple is followed by an EOS.

Enablements can be thought of as pointers to null data. Their only purpose is to facilitate control in scenarios where no explicit data is produced, yet control flow is required. This is especially true when the plan writer wants to capitalize on the state of an operator (again, where no specific data is produced).

For example, when the Compare operator fires, it either reaches a state of TRUE or FALSE. Instead of outputting a boolean object set to the value, it produces one enablement for the TRUE case and a different enablement for the FALSE case. These enablements contain no data (they are self-explanatory), but they facilitate control during execution.

Developers can also define other datatypes and the plan compiler (described later) will enforce type compatibilities between producer and consumer. The requirements for defining a new datatype are

- new datatype must either be per-stream or streamed
- if that datatype is used as an attribute of a relation, it must be “comparable” to another object of the same type (i.e., it must be possible to order the objects)

3.4. Plan Language

The Theseus Plan Language allows authors to specify the following information:

- input accepted by the plan for execution
- data potentially generated by plan execution
- a network of producer/consumer operators describing how data flows through the system
- additional enablements generated by some operators for extra control &

```

PLAN planName
{
    INPUT: planInputType1 planInput1, planInputType2 planInput2, ...
    OUTPUT: planOutputType1 planOutput1, planOutputType2 planOutput2, ...

    BODY
    {
        OPERATOR ( opInput1, opInput2, ... : opOutput1, opOutput2, ... )
        [ WAIT: waitInput1, waitInput2, ... ]
        [ ENABLE: enableOutput1, enableOutput2, ... ]
        [ ON_ERROR: errorOutput1, errorOutput2, ... ] ;

        OPERATOR ...
    }
}

synchronization

```

Generically, plans obey the following structure:

Where:

- *planName* is the name of the plan – it must be unique.
- *planInput* are the names of data objects of that the plan requires as input. The *planInputType* declares the type information for this data
- *planOutput* are the names of output data objects that can potentially be produced during execution.
- **OPERATOR** is the name of any legal Theseus operator (see next section for the operator list). Operators are **NOT** case sensitive.

- *opInput* is the name of data used as input for that operator – it is important the requirements of the operator on input types and order of input types is obeyed.
- *waitInput* is the name of additional data required in order for that operator to execute.
- *opOutput* is the name given to data produced by that operator – it is important to take note of the output types and the order in which they are produced, to ensure correct execution.
- *enableOutput* objects are those enablements produced when the operator begins to execute – “begins to execute” refers to the first time the firing rule on that operator is triggered for a particular transaction. These output enablements, by definition, do not contain data.
- *errorOutput* objects are those enablements produced when/if the operator encounters an error for a particular transaction.

It is useful to describe some additional details related to the various identifiers listed above:

- The *planName* is a unique name for the plan.. The name must be unique because, when deployed in server mode (described later during invocation), this plan name must be distinguishable from other plans. Just as naming is important in any distributed system, so too is it in Theseus. Being able to access a plan from a client front end or via the plan itself (as a subplan, described later) will rely on this feature.
- The *planInput* objects should correspond to input of at least one operator (each) in the plan.
- Some operators accept variable numbers of *opInput* arguments. Plan writers should consult operator documentation before use. All variable argument lists should be appended to the list of those required for input.
- The *waitInput* data is used for operator synchronization purposes. Operators do not execute until all the specified *waitInput* objects arrive (in addition to their own inputs). Normally, *waitInput* objects are non-streaming objects, but they can also be streaming objects. In this case, the operator firing is held up until the first tuple of the relation is streamed. As an example of using WAIT, consider the following plan fragment, where we only want to perform a Retrieve if a previous Retrieve returned any data²:

```

BODY
{
    ...
    RETRIEVE("SELECT *
              FROM Yahoo_Weather
              WHERE temp > 100 and
                    city in ('Los Angeles', 'Anaheim',
                           'Venice')" : hot_spots);

    RETRIEVE("SELECT sun_danger_map
              FROM Weather_COM
              WHERE metro='Los Angeles'" : uv_map)
    WAIT: hot_spots;

    ...
}

```

² **NOTE:** *The Retrieve operators shown in this example are conceptual – they do not correspond directly to the actual Retrieve operators that may have been implemented. Their purpose here is to compactly represent remote queries and their output.*

In the example above, the second Retrieve is only done after the first Retrieve returns any rows. Use of `WAIT` here is both a feature and a convenience. For example, the first Retrieve could have sent its data to an Aggregate operator and then to a Compare operator to check for more than one row (which is all we need to know). Still, this Compare operator would have to have some way of telling the second Retrieve to start – thus, it would also need to post some sort of data-less enablement. Thus, `WAIT` makes this kind of synchronization easy and clean (the data sent to wait just goes into a sink).

- Operators can also generate *enableOutput* data upon execution. This type of data object is the complement of a *waitInput* object – it is produced when an operator has completed execution. For operators that process streaming inputs, this data is produced after the EOS has been processed. *enableOutput* is primarily useful for operators that want to enable another operator upon the completion of their execution, but do not normally generate any data upon output. For example, the Notify operator has no output arguments. By using an *enableOutput* object, the plan writer can trigger the execution of an operator after Notify fires.

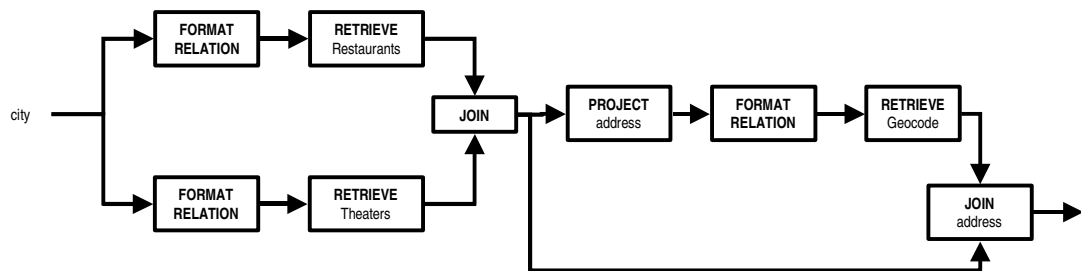
Again, most operators will not need to generate *enableOutput* – their normal output data can be used as a triggering device (via *waitInput* on the operator that should follow). However, for operators that do not produce data (such as Notify) or subplans that do not have any output data, this is a valuable feature of the language.

- The *planOutput* is what can **potentially** be produced by the plan. The plan describes the actions of agent, and if conditional execution is involved, it is possible that not all of the potential data will be produced. However, the *planOutput* lets the plan executor know what data – if any – can be returned back to the caller. In fact, many plans will not have any output – they will produce some side-effect (e-mail, database changes) that does not require data output to the caller.
- Not all *planOutput* need to be consumed – it depends on the needs of the plan. For example, the `Compare_String` operator produces data corresponding to both `FALSE` and `TRUE`. However, if only `TRUE` is actually acted upon, it is the only data that needs to be named. In that case, the operator declared in the plan might look like:

```
BODY
{
    ...
    COMPARE_STRING("thing1", "thing2", "=" : isTrue, _ )
    ...
}
```

where `isTrue` is consumed but the argument corresponding to `FALSE` is ignored. Note that an underscore replaces the unused output that the operator produces.

To illustrate the use of the language in action, and its correspondence to a dataflow graph, we now describe an example plan for TheaterLoc (more examples given at the end of this document). Below we show the dataflow graph for TheaterLoc, along with a sample Theseus plan, written in the plan language. The goal of this plan is to return a list of details (name, address, and geocodes) about the restaurants and theaters in a specific city.



```

PLAN TheaterLoc
{
  INPUT: String city
  OUTPUT: Tuple restaurants_and_theaters

  BODY
  {
    1 FORMATRELATION(city, "city" : restoQuery)
    2 FORMATRELATION(city, "city" : theaterQuery)

    RETRIEVE ("cuisinenet", restoQuery : restaurants)
    RETRIEVE ("yahoo", theaterQuery : theaters)

    JOIN (restaurants, theaters, "" : places)

    PROJECT(places, "address" : place_addresses)

    FORMATRELATION(place_addresses, "address" : geoQuery)

    3 RETRIEVE ("geocoder", geoQuery : geocoded_places)
    JOIN (places, geocoded_places, "address = address" : restaurants_and_theaters)
  }
}

```

In this example, also notice the following (numbered items, above):

- (1) **plan accepts input data:** the conversion of data to a form suitable for Theseus
- (2) **data can be consumed by multiple operators:** no need for multiple signals – adding consumers or producers is simple. In this case both FormatRelation operators consume “city”.
- (3) **plan generates output data:** conversion back to client is automatic

Notice that the above plan uses operators that call functions. These functions are defined in libraries (such as Java class files) outside of the plan code.

3.4.1. Variable Input/Output Arguments

It is possible to have operators that consume or produce variable number of arguments. However, Theseus requires that variable argument lists be grouped, using the “(“ and “)” characters. For example, suppose a Print operator takes a variable number of objects to print. It might be invoked by specifying:

```

BODY
{
  ...
  PRINT((str1, str2) : )
  ...
}

```

Similarly, variable number of output arguments are grouped as well. For example, suppose a Listen operator accepts a series of variable numbers of various datatypes over a network connection and returns that set of data to the plan. It might be expressed in the plan as:

```

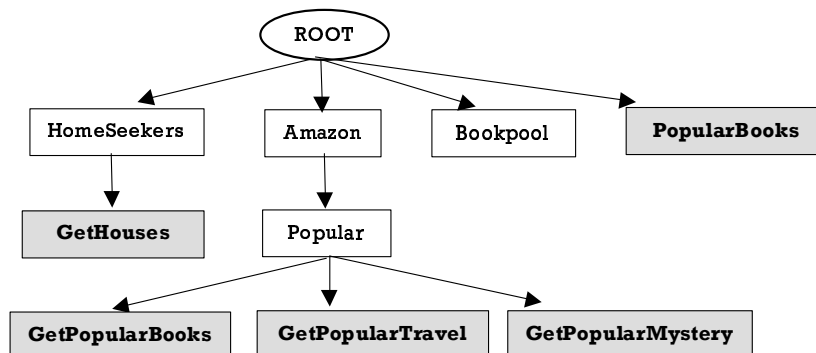
BODY
{
    ...
    LISTEN("hypnos.isi.edu:8080" : (str1, str2, num1, num2))
    ...
}

```

3.5. Naming and Scoping

As described in the previous section, plan naming is an important part of writing a plan. Later, in the section on Invocation, we will discuss how plans are managed, but for now it is important to describe the concept of a plan registry, which has the task of organizing and storing plans for execution. Once plans are compiled, they can be loaded into the registry for remote execution.

The plan registry is very similar to the concept of the registry in most distributed systems or directory in network file systems. It consists of a tree of key/value pairs. The key refers to an entry in the tree. The value will either be NULL (like a folder) or the name of a plan (like a file). For example, the system registry might appear like this:



The boxes above refer to a sample namespace for plans in the system. The filled boxes are actual plans, whereas the unfilled boxes are folders that contain other folders or plans. For example, there is a plan named `/HomeSeekers/GetHouses`. The diagram above also shows plans `/Amazon/Popular/GetPopularBooks` and `/Amazon/Popular/GetPopularMystery`.

Since they are siblings, these two plans are said to be at the *same level of scoping*. That means that they can refer to each other in a relative manner. For example, the `GetPopularBooks` plan might use `GetPopularTravel` and `GetPopularMystery` as subplans and union their results. When calling these plans, `GetPopularBooks` could refer to them directly, without any scoping:

```

GetPopularTravel( : travelBooks)

```

However, notice that there is another plan simply called `PopularBooks`. Suppose the purpose of this plan is to reuse plans in the Amazon and Bookpool hierarchies, so that all of the popular books from these web sources can be easily retrieved. Then, we wish to union them or find the

intersection of their results. When calling the Amazon plan for GetPopularBooks, the general PopularBooks plan must refer to it fully scoped, as:

```
/Amazon/Popular/GetPopularBooks( : books )
```

However, once control has transferred to the GetPopularBooks plan, it can call GetPopularTravel and GetPopularMystery without any scoping qualifiers. This improves the reusability of plans in the system because there is no need for a plan writer to worry how a plan will behave (specifically, relative references to its subplans) if the plan itself is called as a subplan.

Another way to refer to a plan is through relative reference via the “..” string. For example, we could have said:

```
../Amazon/Popular/GetPopularBooks( : books )
```

Some additional notes about plan scoping and the implications of hierarchies:

- Notice that, since plans are scoped, **all data in the system is scoped and guaranteed to be uniquely named**. Since plans must have data that is uniquely named, and since plans must be uniquely named, all data in the system is thus uniquely named.
- The use of a hierarchy easily permits future work in terms of controlling access to plans. For example, the top level of the hierarchy could be indexed by organization and then by user, so that different organizations and users can limit or grant access to various plans in the system. Use of a hierarchy also allows lower levels to enjoy autonomy – organizations can enforce their own access control schemes without having to consult with peers or the root of the system.

3.6. Declaring and Defining Operators

To define an operator in Theseus, a developer needs to do the following:

- Specify the operator metadata
- Write the code for the operator

3.6.1. Operator Metadata

The metadata for an operator contains information about the operator call signature (its input and output arguments and their types) as well as other information about the execution behavior of the operator. Some of this information is optional. Specifically, this metadata is:

- Input arguments and types
- Output arguments and types
- Whether the operator is naturally I/O-bound: = {Yes, No} defaults to No
- Whether the operator is re-entrant: = {Yes, No} defaults to No
- Suggested stream granularity: = {1, 2, .. | “any” } defaults to “any”.

The I/O-bound and re-entrant parameters that allow some implementations of the Theseus executor to scale work resources (worker threads) to meet demand. The stream granularity allows some executor implementations to optimize I/O between operators.

The definition of each operator also contains a database (a registry) of this important metadata. Thus, at development time, the operator developer can specify these important constraints and properties. A non-lethal subset of these (i.e., granularity) can be overridden at the plan and operator instance levels of declaration. However, some properties (i.e., I/O-bound nature) cannot.

Example: the logical set of metadata for a Select operator would be:

```
Operator Select {
    INPUT:    relation InData, string Condition;
    OUTPUT:   relation OutData;
    BLOCKING: false;
    RE-ENTRANT: true;
    GRANULARITY: 100;
}
```

The code for this as an actual operator would be³:

```
public class OpSelect extends Operator
{
    public OpSelect() { }

    // Self-describing metadata
    public OpProperty[] getProperties()
    {
        return new OpProperty[]
        {
            new IntOpProperty(OpProperty.LBL_GRANULARITY, 100),
            new BooleanOpProperty(OpProperty.LBL_BLOCKING, false),
            new BooleanOpProperty(OpProperty.LBL_REENTRANT, false)
        };
    }

    // Self-describing inputs
    public QueueDesc[] getInputs()
    {
        return new QueueDesc[]
        {
            new QueueDesc("in_data", QueueType.STREAMING),
            new QueueDesc("condition", QueueType.NON_STREAMING)
        };
    }

    // Self-describing outputs
    public QueueDesc[] getOutputs ()
    {
        return new QueueDesc[]
        {
            new QueueDesc("out_tuples", QueueType.STREAMING)
        };
    }

    //
    // REST OF OPERATOR CODE
    //
}
```

³ **NOTE:** This code reflects an early version of Theseus API support; it will change slightly upon further integration with the Registry API, however it will be very similar. The key notion is the one of self-describing metadata.

3.6.2. Operator Code

The code for an operator specifies how that operator responds to cases where its firing rule has been satisfied. However, to support for streaming, the Theseus firing rule is more complex than those found in many traditional dataflow systems. This is explained in more detail in the “Execution” part of this document.

The key point to make here is that the burden is on the operator developer to handle all cases where the Theseus firing rule might apply. This is not as complex as it might seem. In general, this means handling cases when the arrival of streaming or non-streamed data at a queue is meaningful. This is discussed in more detail in the Execution section.

4. Invocation

4.1. Methodologies

There are usually two motivations for using Theseus to construct an information agent. One deals with the cases where the user wants a simple way to specify and run an information agent for him/herself. In this case, the user is supplying the input and will use any output like one would use the output of any command-line stream (i.e., such as the output of a UNIX command, like “ls”). Another motivation for using Theseus is one where an administrator wants to “deploy” a Theseus plan so that it can be accessed by multiple users for concurrent, scalable, high-performance plan execution. The modes of access represented by these two scenarios represent can be summarized as the desire to execute Theseus via either:

- **the network:** plans run like application servers (waiting for requests)
- **the command line:** plan run within a user shell, like most executable programs

4.1.1. Invocation via Network (Server Mode)

For V2, it is assumed that the norm will be to run Theseus in “network mode”. The motivation behind this is similar to why application servers have become popular:

- Many clients want access to similar functionality – they only differ in the types of data arguments they send to the system.
- A back-end system usually runs on a better hardware/software platform and can achieve a higher level of performance than could deployment on the average user platform.
- Limited-capability devices, such as PDAs, cell-phones, up-and-coming network appliances will likely have network access but will only contain limited amounts of memory and local persistence. These devices need to be able to access servers for complex functionality and use the network as their interface.
- Use of a common plan can result in savings not possible when the plan is run independently by users (caching, data locality issues, better bandwidth, etc).
- It is easier to access an application over the network than it is to (a) download it to your own machine, (b) struggle with installation, and (c) keep track of software updates.
- Other modules that the caller does not or should not know about (in the case of Theseus, this means subplans) are not required to be local. This is the same justification/appreciation of the transparency realized by most distributed systems.

There are also some Theseus-specific motivations for server-mode access:

- the monitoring aspect of Theseus is such that some sort of executor is “alive” constantly, waking up to execute at the right time (the opening of the stock market, etc).
- some of the requirements necessary for information management (i.e., use of an external database) would otherwise require additional complicated downloads and expense by the user
- certain optimizations (i.e., speculative execution) benefit from execution being an ongoing process

Given these requirements, the goal of server-mode Theseus is to enable clients to connect via the network to an always-available, scalable back-end system. Optimization of Theseus in this style of deployment focuses on improving parallelism and concurrency.

4.1.2. Invocation via Command Line (Interactive Mode)

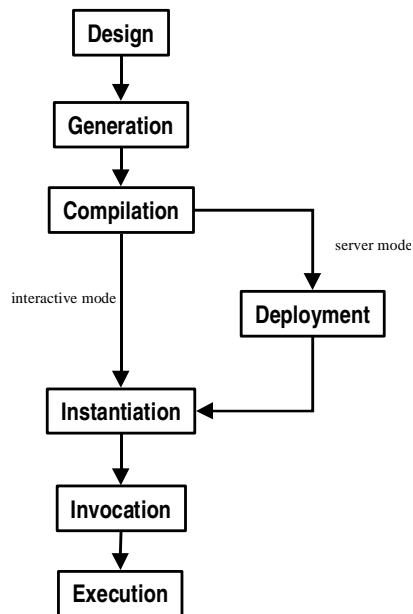
Although server-mode will be a popular style of deployment, it is also reasonable to allow Theseus to execute plans within a user shell. In this scenario, data is fed to Theseus via the user command shell and output data is returned to that same shell.

Note that the concept of transactions in interactive mode is still relevant: since a plan may call subplans recursively, there is a need to distinguish between different frames of that recursion.

4.2. Preparing For Invocation

4.2.1. Plan Lifecycle

Plans will typically follow the lifecycle shown below:



These stages are defined as:

STAGE	DESCRIPTION
Design	Specification of a plan as an idea by the designer.
Generation	The production of plan code in the Theseus Plan Language - either via editor or GUI.
Compilation	Semantic checking, local static optimization, insertion of additional adapter operators to facilitate execution, creation and output of a persistent plan data structure.
Deployment	Loading of plan into repository, so that it can be instantiated via the network. Global static optimization possible at this phase.
Instantiation	Realization of a plan at runtime: the conversion from a persistent data structure to real resources (i.e., from data structure to real runtime threads)
Invocation	Presentation of coordinated set of input to a plan instance. The coordinated data at this point is tagged with a unique transaction identifier.

Execution	The resulting consume-produce-consume cycle of dataflow processing induced by the initial input. Output may be generated sporadically during this phase. Termination may also occur. See Section 5 for more details. Dynamic local and global runtime plan optimization is also possible at this phase.
------------------	---

Some of the above phases are self-explanatory (Design, Generation, etc). Others are not, but before covering these more complex phases in detail, we first briefly introduce what plan optimization means in Theseus.

4.3. Plan Optimization

A subtle point about the phases above is those stages at which plan optimization is possible. *Optimization* is defined as the process of improving plan execution, either in terms of performance, scalability, resource use, or reliability. Optimization can be done for either *local* or *global* reasons. Optimizations may also be *static* or *dynamic*, depending on when and how they are applied. We now describe four examples that demonstrate the possible types of optimizations and at what phase in the plan lifecycle they are possible.

For example, “pushing Selects” is the process of identifying Select operators in a plan and moving them in front of other relational operators so that data filtering may occur earlier in processing and thus make execution later in the flow less taxing (fewer tuples to process for later operators). This is a static local optimization because it occurs at compile time and because it occurs on behalf on a single plan.

Another type of optimization involves the concept of plan merging. As multiple plans are deposited into the plan repository, similarities between them may motivate the merging of plan subgraphs. That way, when deployed, multiple plans may be able to share operators (and thus conserve resources) or data gathered by one plan might be used in others (rather than repeating the retrieval) . This is an example of a static global optimization because it occurs in the deployment phase (when plans are deposited into the repository) and because it involves the analysis of some or all of the plans in the system.

A third example optimization is that of dynamically scaling the number of worker threads for a given operator. This might be useful when a deployed plan receives a substantial load of requests. The number of operator worker threads might need to increase to meet the demand. This is an example of a dynamic local optimization because it is done during the execution phase but it affects only the local plan.

Finally, an example of a dynamic global optimization is that of migrating deployed plans to different plan servers on the network. This would be useful when, say, two plans are competing for a single CPU. Based on the demand for these plans, the plan server may re-route the deployment of one of the plans to another, underused machine. This is thus an optimization that is done at execution time, but is based on run-time characteristics.

4.4. Plan Compilation

Plan compilation is the phase where many static local optimizations will be performed. For now, the only two purposes of compilation are to (a) represent the Theseus plan in a *instantiable format* and (b) to insert the default error-handling operators. Instantiable format really refers to a persistent data structure that represents the system as it is ready to execute a specific plan. This is easily possible using today’s current programming language technology. For example, it is possible in Java to represent the system as a ThreadGroup and then to persist that data structure using the Serialization facility.

Although error handling is covered in more detail in a later section, it is enough now to say that the Theseus compiler at compile-time will insert error-handling operators (the Void and Restart operators).

4.5. Plan Instantiation

At compile time, it is minimally necessary to augment a given plan for purposes of supporting invocation. This means inserting special operators into a plan. These operators are not pasted into the source code for the plan, nor do they exist in the compiled form. They only exist upon plan instantiation.

4.5.1. Input and Output Adapters

To facilitate invocation, it is necessary to insert *input and output adapter operators* that help convert incoming data and outgoing data to forms that are suitable for Theseus and the client, respectively. Inserting these operators makes the rest of plan instantiation a generic process. The style of input and output adapters depends on the invocation tool used. Abstractly, there are two types of interfaces to plans: the user shell (command line access) and the network (server mode).

If a plan is instantiated from the user shell, **input adapters are inserted that read input data from standard input or from a file**. If the plan is served over the network, **input adapters are inserted that can read data from the network** over a specific socket (or set of sockets), as directed. Later, we will describe the Plan Service, but for now it is enough to say that this service has knowledge about all plans being served and thus it can choose socket identifiers that do not conflict with input for other plans.

Recall from Section 3 that plan input types are declared by the plan writer – it is these types that determine how input data is converted. For example, if a plan requires a String and Number as input, as shown below

```
Plan MyPlan
{
    INPUT: String MyString, Number MyNumber
    ...
}
```

Input adapters will ensure that incoming data is converted/casted to the types specified.

Input and output adapter operators cannot be inserted into the plan prior to instantiation, since their very nature (how they marshal input and output) depends on the method of invocation. For example, interactive mode invocation means that the input adapter needs to grab input data from the command line, not from the network (as would be true in server mode).

4.5.2. Input and Output Formats

There is also a great degree of flexibility in how input may be specified by the caller. In the above examples, the name-value string based method is used. This style is one where the caller just specifies data arguments as a named series of strings and they are converted into the proper Theseus type. However, there will be at least one other input data format: XML.

The XML input will likely be a popular type of input and we cover it in more detail here because it represents a good example of how data is labeled and how input can refer to a common session or distinct sessions. More specifically, input sources must include information on

- where the plan input starts
- where one or more sessions start
- the labeling of the data

For example, the input to a plan might be contained in a file called `INPUT.XML` and that file

```
<XML>
<PLAN_INPUT>
<SESSION>

<RELATION>

<NAME>
People
</NAME>

<ATTRIBUTE_LIST>
<ATTRIBUTE>Name:String</ATTRIBUTE>
<ATTRIBUTE>Age:Number</ATTRIBUTE>
</ATTRIBUTE_LIST>

<ROW>
<CELL attr=Name>Jason</CELL>
<CELL attr=Age>25</CELL>
</ROW>

<ROW>
<CELL attr=Name>Jane</CELL>
<CELL attr=Age>47</CELL>
</ROW>

</RELATION>

<CONDITION>
Age > 35
</CONDITION>

</SESSION>
```

might look like this (assume the input drives a single `SELECT` operator):

In fact, the above input is currently the same type that is supported by the default Theseus V2 XML client `xmlcli`.

Allowing the XML input format also opens the door to running plans stored on web servers across the Internet. Instead of clients submitting requests to run plans, Theseus could theoretically be notified of URLs that contain plan inputs and then read the XML-input from those network addresses.

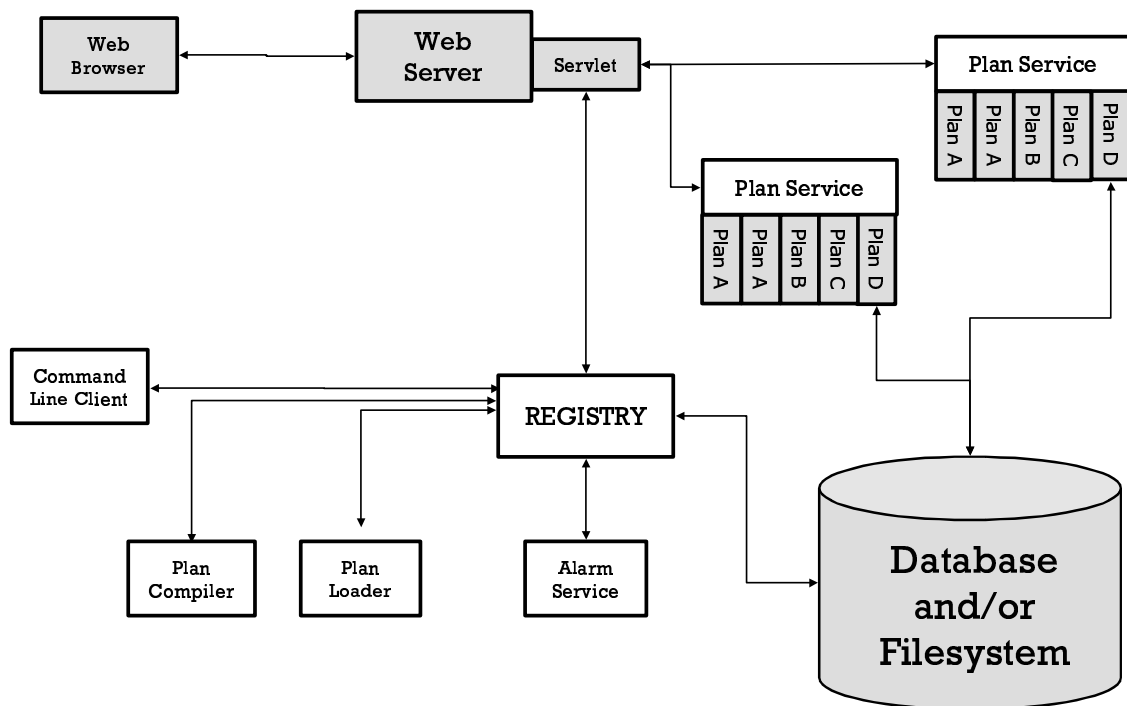
Unless otherwise specified, the plan output format is based on the plan input format. If XML is used to invoke the plan, XML is returned.

5. The System

The previous two sections abstractly described Theseus plans and how they are invoked. Both made reference to a few components that were assumed to exist. In addition to these, there are some other key system components important in the enterprise-level deployment of Theseus. Specifically, these components are:

- plan compiler
- registry
- command-line client
- plan service
- alarm service
- plan loader & unloaded
- external database

A view of the system in terms of these components can be seen in the figure below:



5.1. Components, Tools, and Services

5.1.1. The Registry

For high reliability and automatic configuration, Theseus includes support for a Registry service. The Registry acts as a repository for all components in the system in which key configuration and run-time data is stored, including:

- default system configurations (i.e., plan path – directories where plans are stored)
- default component configurations (i.e., default logger settings)
- existence of components (i.e., hosts & ports of running plan servers, databases)
- local plan configuration data (i.e., logging level for that plan)
- global plan configuration data (i.e., hierarchy/organization of all plans)
- operator properties (i.e., default SMTP host for the Notify operator)
- system information (i.e., Theseus version)

The registry organizes its data hierarchically, like a directory structure in a filesystem: as a tree of nodes. Each node may either be a folder or an entry. An entry is a name value pair. An example of the organization and types of information the Registry will store is shown below:

```
+ System
  Version = 2.1.3

+ Environment
  THE_PLAN_PATH = c:\my_plan_path

+ Components
  + TS_Logger
    Default_Log_Mask = ERROR | WARNINGS | TRACE

  + TS_Executor
    ...

+ Operators
  + Join

+ Plans
  + TheaterLoc
    Plan = d:\plans\theaterloc
    Max_Users = 100

  + PopularBooks
    Plan = d:\plans\pop
    Log_Mask = ERROR | WARNINGS

  + PopularAmazonBooks
    Plan = e:\janes_plans\amazon

  + PopularBordersBooks
    Plan = e:\joes_plans\borders_books
    Instances = 2
  + Instance_1
    Host = brawn.isi.edu:8031
  + Instance_2
    Host = machine.usc.edu:8044
```

Scalability is not really an issue when it comes to the registry. For one, there can be multiple instances of the Registry on a host. Second, the data the Registry persists does not change that often. Third, plan specific registry nodes are communicated to the plan when it is invoked. The runtime executor treats this data as a write-through cache. If multiple registries are running, their consistency and ACID-like properties will be guaranteed (in V3, likely).

One key feature of the Registry is its ability to act as an interface for dynamically changing system and plan properties. For example, suppose 10 instances of the TheaterLoc plan are running on the system. When these plans were initially served, suppose the default logging level for TheaterLoc was “ERROR” (meaning: log only error messages). This was declared by having an entry called LOGGING_LEVEL in the TheaterLoc plan node. Also suppose that these plans are fairly busy managing transactions. If an administrator wanted to change the logging level of all TheaterLoc plans (or just the logging level of a specific instance) so that full tracing was turned on, he could update the registry by contacting the registry and resetting the LOGGING_LEVEL variable to “TRACE”. The Registry would then contact the running TheaterLoc plans and notify them of this configuration change. Thus, the logging level of running plans could be changed without having to bring them all down and restart them. To be notified of a relevant change in the Registry, system components register callbacks with the Registry.

The Registry is the only service that a user needs to know about when running a plan. Users simply set an environment variable pointing to that Registry. When running any of the Theseus runtime components (command line client, etc), this environment variable is used to locate the Registry.

The Registry persists its state in a database. Thus, if the Registry is inadvertently killed, it can reset itself to the last set of prior key mappings.

When the Registry is started, it assigns itself to the port matching that of the environment variable contained in the shell of the user who started it. For example, if my THESEUS_REGISTRY environment variable is set to hypnos.isi.edu:8004, starting the Registry assigns that registry to port 8004 on hypnos.

5.1.2. The Alarm Service

The Alarm Service is responsible for running pre-scheduled plans. When the Registry is started, it checks to see if any alarms have been set. If so, it starts up the Alarm Service and notifies that service of the list of alarms. Additional changes to alarms are communicated from the Registry to the Alarm Service. When there are no more alarms set, the Alarm Service (if running) is shut down by the Registry.

Pre-scheduled plans can have input data and a target e-mail address associated with them. When the Alarm Service runs a plan, it seeds it with this input data. For example, one could pre-schedule a TheaterLoc plan to run at midnight every night, using “Santa Monica” as its input. When the plan completes, it delivers notification to the e-mail address specified, including the results of any output data generated. In the future, if demanded, alternative notification schemes will be supported. Obviously, the plan writer can construct these manually, so there is not a pressing need to support flexible alternatives – just a default behavior.

5.1.3. Command Line Client

There is a command line client that allows plans to be run from the user shell. To run this client and use the other parts of the system (Registry, database, and other components), the user should have his THESEUS_REGISTRY variable set appropriately.

Plan input data is communicated through standard input of the shell. This input can be encoded using a variety of protocols and standards. The default is XML.

5.1.4. Web Server (HTTP Server)

There is also a default network (HTTP) server that accepts HTTP POST requests for plan invocation. Users can interface with this client in any number of ways, the most likely being just writing a web form that, upon submission, performs an HTTP POST of the form contents. In addition to plan input data, the name of the plan is also specified in the POST request.

Typically, one would write a servlet or CGI-style program to interface with the Plan Service(s).

5.1.5. Plan Compiler

The Plan Compiler converts plans written in the Theseus Plan Language into a run-time data structure that can be persisted. As explained earlier, the process of plan compilation takes a plan as input and generates a run-time data structure as output. The compiler also performs static local optimizations.

5.1.6. Plan Loader and Unloader

The plan loader performs two functions: (a) it adds/updates a compiled plan to the system database (for flexible retrieval) and (b) it adds/updates the entry for that plan in the plan naming hierarchy. The advantage of (a) is that plans can be requested from the network without having to worry about read/write access to the local filesystem. The process of (b) is to maintain the naming hierarchy (described in Section 3.4) that is important when running subplans or otherwise referring to other plans.

The plan unloader removes a plan from the database and the hierarchy.

5.1.7. External Database

Both the Registry and individual plans will make use of external databases for state persistence. The Registry will store its name/value hierarchy in the system database, while plans will use either the system database or a specific, custom database, for information management purposes. For example, when the HomeSeekers plan wants to “remember” what houses it has seen before, it will use an external database to store this information, since the HomeSeekers plan may not be constantly running (and thus could otherwise not remember this set of seen houses in between sessions).

5.1.8. Plan Service

The Plan Service simply hosts a set of plan instances, each of which listens on network ports for input data. For example, a plan service might host 10 instances of a TheaterLoc plan or it might host 5 instances of a TheaterLoc plan and 1 instance of a HomeSeekers plan. There can be multiple instances of plan services. Registry decides when to load a new plan service. The plan service allows plans to accept network connections as input. The data that comes on these network connections can obey any one of several flexible protocols (XML, HTTP, etc).

5.1.9. Plan Scheduler

Allows plans to request periodic execution. As input, the scheduler requires a plan and start date (optional), a stop date (optional), and a frequency qualifier (required). Plans can also be unscheduled with this tool.

5.2. An Important Note About the Components

As the reader will no doubt have discovered, some of the services have dual roles and there is some lack of specification in various places. For example:

- the Registry is responsible for starting up and shutting down the Alarm Service, just like a Lifecycle service usually does in other component architectures
- the Plan Service acts a name server as well as application server
- the plan for coordinating multiple registries is hand-waved

More detailed, complex architectural specifications have not been included here because the Theseus platform is only being introduced in V2. The components listed are the chief components of the eventual Theseus platform. However, the scalability of the components themselves (as well as isolating their roles) is a task left for V3. Instead, the purpose of V2 is to focus on the scalability and performance of the plan execution system.

5.3. Examples of Using The System Components

Here we provide several examples of how the components described above are used to accomplish plan running, serving, and invocation.

5.3.1. Interactive Execution, Assuming Nothing

In this case, we assume that the developer just wants to write and run/test the plan. No environment variables are set, etc.

The steps required are:

- Developer writes and compiles plan (Plan Compiler tool)
- Developer runs named plan (Command Line Client)
 - Command-line client looks for Registry but fails (no env variable set)
 - Command-line client creates its own registry (as a thread)
 - Command-line client locates compiled plan in file system
- Plan executes and terminates

5.3.2. Interactive Execution for Deployed Plan

Suppose a developer writes a plan and wants to deploy it so that it can be shared or globally optimized (plans in the database could be globally optimized using plan merging, etc).

In this case, we assume that only the following steps have already been completed:

- ❖ Database has been started
- ❖ Registry has been started
- ❖ User has environment variable that correctly points to the registry

The steps required are:

- Developer writes and compiles plan (Plan Compiler tool)
- Developer deploys plan (Plan Loader tool)
 - Compiled plan is inserted into the database
 - Registry plan name hierarchy is updated
- Developer runs named plan (Command Line Client)

- Command-line client looks for Registry, finds it, initializes
- Command-line client asks Registry for named plan
- Registry locates plan in database, returns it to client
- Plan executes and terminates

5.3.3. Serving a Deployed Plan

Serving a plan allows it to be accessed via the network easily, with minimal overhead.

In this case, we assume that only the following steps have already been completed:

- ❖ Database has been started
- ❖ Registry has been started
- ❖ User has environment variable that correctly points to the registry

The steps required are:

- Developer writes and compiles plan (Plan Compiler tool)
- Developer loads compiled plan into the system (Plan Loader tool)
- Administrator requests that plan be served (Plan Server tool)
 - Registry is queried by plan server tool to see if any plan service is running
 - Registry responds that this no plan service exists – one is started
 - Registry responds with address of plan service
 - Plan server tool requests that plan service run specified plan
 - Plan service loads plan from database, notifies Registry

5.3.4. Invoking a Served Plan Remotely

Again, in this case, we assume:

- ❖ Database has been started (but not local to caller)
- ❖ Registry has been started (but not local to caller)
- ❖ Web Server and Plan Service Interface Servlet exists on target machine

The administrator steps are:

- Administrator writes Plan Service Interface Servlet, which needs to do the following:
 - During initialization, successfully contact the Registry
 - Download list of running plans and their addresses
 - Sets callbacks with Registry to be notified about plan service modifications

The user steps required are:

- Developer uses makes HTTP POST to Plan Service Interface Servlet
 - POST contains name of plan and input arguments
- Servlet sends request to Plan Service
- Plan Service converts input data to proper format, invokes proper plan
 - Results are captured and sent back to servlet
- Servlet reformats these results in HTML and returns result

Notice that what is described above is the general case whereby any plan could be invoked. It is also obviously possible to write a servlet./CGI/whatever that is dedicated to interfacing with a particular plan.

Also note that the Plan Service Interface Servlet registers callbacks with the Registry so that if any new plan services are run – or if any running ones are stopped – it knows about it and can thus route appropriately.

6. Execution

Theseus functions as a virtual dataflow machine. At runtime, the *execution* of a plan refers to the routing of data between a network of operators, just as data is routed along arcs between actors in dataflow computers. The main difference is that Theseus is implemented in software, whereas traditional dataflow computers are implemented in hardware. In this sense, Theseus is much closer to the current trend of hybridization in dataflow systems – specifically, the implementation of dataflow actors as threads running atop a von-Neumann architecture.

Upon execution, all of the operators in a Theseus plan are *instantiated*. The details of what operator instantiation means are important, but they are covered later in this paper. For now, it is enough to say that every operator is available at the beginning of execution, so that – in the absence of resources – complete execution is still possible. That is, there are at least enough resources to process a single transaction. This is in contrast to the instantiation-on-demand style of some dataflow systems, in which it was possible that deadlock could occur due to lack of resources.

Operators in Theseus act as state machines. They take some set of input arguments and, based on these, achieve some state. Although operators are instantiated at the beginning of execution, they do not *fire* until their firing rule has been satisfied. Firing is the process of consuming some combination of named input objects for the purposes of (a) producing output or (b) evolving internal state (building one of the relations in a pipelined style Join – no emission of tuples). The firing rule in Theseus is more complex than it is in typical dataflow systems, and the details of this rule are discussed later in this section. For now, it is enough to say that operators fire when they have sufficient input. Upon firing, operators may produce output. This output is named and subsequently consumed by operators that require those named objects as input.

Upon invocation, all incoming data is labeled with a transaction identifier (transaction ID). This identifier is discussed in greater detail later, but it is enough to say that this ID allows operators that require multiple inputs to match proper data objects together. In the case of guaranteed single invocation, this is normally not necessary – data is obviously all related to the same logical invocation. However, for concurrent sessions, it is necessary to distinguish data from each other. Furthermore, recursion in Theseus is such that operators need to be able to distinguish levels of recursion from each other – these are also components of the transaction ID. The use of the transaction ID in Theseus is directly analogous to the use of tags in dynamic dataflow systems.

Since operators fire independently of each other, it is usually the case that execution will occur in parallel. As discussed later, whether this parallelism is “real” or “virtual” (interleaved threads) depends on whether the underlying OS and hardware platform supports (a) real threads and (b) has more than one CPU (i.e., SMP or cluster environments). In addition to parallel execution, operators produce data independently of when it is consumed – that is, the communication of data between operators is asynchronous. To achieve this asynchrony, operators maintain *queues* on which data is *pipelined*. Thus, an operator does not need to process data immediately as another operator produces it. Instead, it processes it when it has the resources – in the meantime, the data remains on the queue.

Theseus plans terminate when all non-static operator inputs (non-static operator queues) are empty. That is, no running transactions exist. In this way, plans may contain operators that have static arguments without worrying that the presence of these static arguments will inhibit termination.

6.1. Summarizing Execution

To summarize, independent of how invocation is achieved, here are the basic steps of execution:

1. Plan invocation occurs: a set of named data objects is sent to a specified plan in a distinct format (string, XML, etc).
2. This incoming data arrives at an input adapter operator and is converted/casted to a form suitable for internal processing in Theseus and is labeled with a unique transaction ID.
3. Operators that consume that data fire as appropriate, based on how the firing rule applies to them – thus, based on the type of input arguments they accept. Operators match data by transaction by ID in order to determine when a coordinated set of input has been received.
4. As operators fire, they may evolve some internal state and/or they may produce data. If data is produced, it is labeled with the same transaction ID as that of the input data that induced the firing.
5. Operators that subsequently consume this data do so asynchronously.
6. The produce/consume process continues asynchronously and in parallel.
7. Data that is declared as “plan output” is sent to the output adapter operator, which formats it for return to the caller based on the input format and the method of invocation.

The following sections explore the execution style described above in greater detail.

6.2. Streaming

Streaming plays a fundamental role in Theseus execution. As described earlier, it allows output data to be generated as soon as possible, exploits parallelism better, and improves performance for I/O-bound plans. In this section, we describe the details of streaming and solutions to some of the issues that it raises.

First, we define streaming. In Theseus, a streams are a series of tuples, concluded by an *End-of-Stream* (EOS) that belong to a single, logical relation. Streaming is useful because it allows operators to process at Tuple-level granularity, rather than Relation-level granularity. The EOS is important because it allows operators to know when a Relation input has concluded. This is particularly important when the operator maintains some sort of internal state, such as that necessary for Sort or Join. With these operators, output of tuples may be halted until some or all of the input relations present the operator with a corresponding EOS. In such situations, once this token is received, output processing can begin.

When an operator receives only an EOS (no tuples), it is assumed that the relation simply has no data. This can happen in many cases, most commonly when all of the data being processed is filtered out (i.e., after a Select operator).

The Theseus Plan Language allows operators to declare the data types of their input and output. Streaming only occurs for the Relation data type. Non-Relation data types (String, Number) are not streamed. Furthermore, if an operator contains both Relation and non-Relation input arguments, then the non-Relation arguments are processed on a *per-stream* basis. Specifically, these non-Relation arguments correspond to a given stream. Finally, if these non-Relation arguments are declared statically in the plan language, they apply to *every stream*.

Optimizing operator execution in the presence of streams is the responsibility of the operator developer. Clearly, there are times when per-tuple processing is useful and when it is not. For example, processing tuples at a fine granularity is appropriate for an operator like Select, which operates very fast and can output data almost immediately. In contrast, an operator like Db-Insert, which inserts rows into a table in an external relational database, is sub-optimal if it executes on a per-tuple basis. Although it may be streamed one tuple at a time, this operator could take advantage of the state maintenance (see below) facilities of the system to wait until several tuples have been acquired before initiating a remote database insert.

Altering the granularity of tuple processing, in the presence of streams, is therefore left to the discretion of the operator developer. However, there is also a role for the system to perform a type of dynamic local optimization here: the system could monitor the queuing behavior of the various operators and adjust their granularities dynamically to reduce the overhead of fine-grain tuple passing when it is of no benefit (in cases where consumer queues are very full, implying a slow consumer operator).

6.3. Transactions

In Version 2, Theseus labels each execution (the input, the intermediate results, and the output) with a logical transaction ID. The purpose of this ID is thus to distinguish data from each other in a concurrent execution environment (such as a plan server), to distinguish recursive stack frames from each other, and to also provide an index to a logical set of data active in the system at any one time. This last, more subtle purpose is to allow a hook for the system to treat the distributed data as a logical whole. For example, if a transaction needs to be voided, the system has a convenient index on which to locate all data for that transaction, distributed among operator queues. Use of a transaction ID in the case of recursive stack frames is covered in more detail in the “Subplans” part of this section.

6.3.1. Transaction Management

Transaction management is completely transparent to the operator developer. He/she need not worry about mixing up data between transactions, since the operator itself plays no role in the assignment or management of transaction IDs. Operators will be executed when enabled by data belonging to a logical transaction.

We now describe how transaction IDs are created and propagated during execution. Plan input data is labeled when it enters the system with a unique transaction ID. In turn, when operators execute on that data, they each label the corresponding output with the same transaction ID. This process continues with all operators that execute – since they are made aware of the transaction ID when they execute, they are able to label the output data appropriately.

Again, it is important to emphasize that the operator developer need not need worry about these IDs. When operators are invoked, they are supplied a pointer to a data structure containing metadata about the invocation, which includes the transaction ID. For all system-level functions (such as state maintenance, getting data from input queues, setting data on output queues), the operator supplies this opaque object to the system and the system does the rest. The developer is assured that processing, state management, and the input/output of data is for a logical transaction.

6.3.2. Transaction Lifetimes

Dataflow processing enables significant parallelism and performance through its decentralized approach. Dataflow actors are autonomous – when data is available, they process it and produce results. As described earlier, the operators in Theseus are directly analogous to these

actors. However, Theseus operators are more complex in two ways: (a) they can maintain state and (b) they are not as fault-tolerant as typical dataflow actors because they rely heavily on external resources (i.e., the Web, external databases). Further complicating things is that Theseus is interested in supporting concurrent user invocations, a consideration not frequently addressed in dataflow literature.

When external resources fail, it may cause a plan to only complete half of its processing (the rest never being enabled because the resources were unavailable). At this point, the plan itself contains some global state in the sense that there are operators managing data on behalf of a transaction that has failed. Although the transaction management mechanism ensures that data from a different transaction will not be confused with the data from the “half-baked” transaction, there still needs to be a mechanism for voiding a failed transaction.

One method for voiding a transaction is to create an operator that does it explicitly. This approach is clean, but it assumes that (a) an operator was developed for this purpose and (b) the plan author has made sure to invoke this operator when any possible error occurs. Specifically, the lower-level system really does not have any guarantee that the higher layers of the system will clean up a failed transaction adequately. As an additional safeguard, the system supports the notion of transaction *lifetimes*, implemented as a simple form of timed-to-live (TTL) data.

System administrators can set the default TTL for transactions in the system so that operator queues can cleanse themselves of stale data. The strength of this approach is that it is clean, guaranteed to work for all operators, and completely decentralized. The only weakness is that it the administrator may specify too low of a TTL for some plans.

6.4. Operator Firing Rule

All dataflow systems specify some type of firing rule obeyed by actors (operators in Theseus) during execution. In static dataflow systems [Dennis 1974], that rule is

execute when an operator has tokens available on all of its input arcs

In tagged-token, dynamic dataflow systems [Arvind et. al. 1981], that rule is:

execute when an operator has tokens that all have the same tag available on all of its input arcs

In Theseus, the firing rule is

*execute when an operator has, for a particular **transaction**, tokens available at all of its non-streaming inputs and any of its streaming inputs*

Note: the Theseus notion of a transaction is essentially a different word for the dynamic dataflow concept of “tokens that have the same tag”. In both cases, the idea is the same: to support concurrency and recursion in a dataflow system, there must be some mechanism to allow the system to distinguish groups of inputs from each other.

The reason for having a firing rule that is slightly more complex than that of standard dynamic dataflow systems [Arvind et. al. 1981] has to do with the nature of the plans Theseus typically executes and the operators usually involved. Although Theseus can function as a general dataflow system, it is particularly optimized to execute information agent plans. More specifically, Theseus plans usually retrieve and integrate relational data from multiple sources and then process or manage that data in some way. Also, many of the operators in Theseus (Join, Select, etc) are such that they can actually perform useful work even when *only parts of some* of their relational inputs are available. For example, it is possible in a dataflow environment to perform a pipelined Join, as described by [Wilschutt et.al.1991].

One of the implications of allowing operators to execute in the presence of partial inputs is that state must often be maintained between executions for a logical transaction. For example, when performing a pipelined Join, we must keep track of *inner* relation until we see the EOS from the *outer* relation. During execution, the arrival of outer relation tuples is used to probe the inner relation. [Ives et. al. 1998] described a pipelined join similar to that which is possible with Theseus – one that is adaptive in the sense that the inner or outer is determined based on the arrival of the first EOS.

Below are some examples of how Theseus operators obey the firing rule described:

Example 1 (one streamed, one non-streamed): The Select operator takes two inputs, a relation (the *data*) and a string (the selection *condition*), and outputs a *filtered* relation. This operator will only fire for each tuple of data that arrives, but only after the selection condition for that transaction has also arrived. Thus, if the condition has not arrived, the tuples of data will simply queue (no execution being triggered) until that condition arrives. Every time it fires, this operator can generate result tuples.

Example 2 (both streaming): The Union operator takes two relational inputs, a *left* and a *right* relation, and outputs a *unioned* relation. This operator fires whenever a tuple on the left or right arrives, outputting tuples of the unioned relation as necessary. This operator also maintains state – it keeps track of the tuples it has seen, so that no duplicate tuples are produced.

Example 3 (both non-streaming): The CompareString operator takes three string inputs: the first two being treated as data and the last being an comparison condition. Based on the condition and the data, the operator outputs a token on either its true or false enablement output queues. In this case, there are no relations, so the operator does not fire until all of its string inputs have arrived.

6.5. Operator Processing

While the firing rule is clear about *when* the operator executes, it does not say much about *how* it executes. For example, suppose a Select operator receives its selection condition first and then eventually receives its first tuple. How are both inputs processed? Also, what happens if the Select operator receives its first tuple and selection condition at exactly the same time?

In general, the firing rule is activated by either (a) producing operators notify consuming operators and/or (b) consuming operators monitoring their queues. The details about which approach (or combination of approaches) is left up to the implementation of the executor. In any case, once the firing rule applies, the relevant inputs must be processed.

In gathering the inputs to be processed, the executor ensures that the inputs are from the same transaction. Also, if an operator consumes any streaming inputs, the EOS for each of these streams is processed separately. The gathering of inputs that obey these constraints is referred to as *gathering work*. More specifically, when an operator knows that, for a particular transaction T, the firing rule has been satisfied, it does the following:

- 1 Gets data from all queues corresponding to T , including the EOS markers, if any
- 2 Foreach per-stream input p
 - 2.1 Add input to data structure that contains all per-stream inputs.
 - 2.2 Call well-known function (each operator has this function) with this data structure as an argument
- 3 Foreach streaming input s
 - 3.1 Call operator function for processing tuples on s
 - 3.2 If there was an *EOS* on s
 - 3.2.1 Call the operator function for processing the *EOS* on s

To understand why this algorithm works, it is useful to consider a few examples. In each of these, we will consider the behavior of the Select operator. Suppose that it consumes both a Tuple T containing $t_1..t_n$ tuples (data to be filtered) and a string S (selection condition). Note: the word “process ...” below is used as an abbreviation for “call the operator function to process ...”.

For the first case, suppose **some** of TS (tuples $t_1..t_k$, where $k < n$) is sent to Select before S . In this case, the operator will wait until it receives S . At that point, it will collect $t_1..t_k$ and S . Then, it will first call the well-known function for processing all per-stream inputs (S is the only one). This operator will copy this data to its local state. Then, the tuples will be processed. They will be able to be filtered because S was stored before the functions for processing the tuples were called.

As a second example, suppose S is received first and then $t_1..t_n$ is received all at once. In this case, as in the previous, the operator will not fire until at least t_1 and S are present. However, since all of the tuples have arrived, as well as S , the operator will gather all of them and do the following: process S (save it to local state), process $t_1..t_n$ tuples, process the EOS for T for that transaction. After processing the EOS, the system will automatically discard S because the stream has finished. If S had been static data, it would not have been discarded.

One additional point to make is that writing operators will require that these processing functions are also coded. The system could theoretically enforce this, although it is not clear if this will be enforced immediately in V2. The instructions for developing an operator conforming to the needs of the algorithm above is contained in the Implementation section “Developing Operators”.

6.6. State Management

In Theseus V2, state management has to do with the maintenance of data by an operator relevant to a particular transaction beyond the duration of a particular firing. The reasons why state management is necessary include:

- **streaming:** execution happens at the tuple level, so to implement things like pipelined Join, we need to maintain state (the probed queue, for example, in the pipelines Hash Join case)
- **multiple transactions:** operators need to be able to executing concurrent pipelined Joins – that is, pipelined Joins for two different transactions can still fire in an interleaved fashion. Thus, some sort of transaction-indexed state management will be necessary.

- **optimization of data passing:** it is possible to implement stateless operator by continually feeding back to the operator the stateful information. However, this leads to an enormous amount of message passing

Operators maintain state using specific system APIs for getting/setting stateful data structures. Each operator implementer designs the contents of the class that will be persisted by the manager at runtime. During operator execution, these APIs:

- allow the state to be loaded **efficiently** (i.e., just a pointer is returned)
- handle transaction indexing **automatically** and **transparently** – the caller does not need to know anything about what transaction is occurring, just that it is occurring

Although we do not specify the state management API here, it seems important that, minimally, the following be permitted:

- locking and unlocking of state
- accessors/mutators of state

6.7. Subplans

Subplans are a mechanism for allowing one plan to call another during execution. The caller is referred to as the *parent plan*, while the callee is the *child plan*. The Theseus plan language and execution system have been designed to support subplans. In terms of language, referring to a plan by its name in a plan allows a plan to execute like any normal Theseus operator. When the required data is available, it is fed to the child plan. Data returned by that plan is treated just like the output of an operator – it is broadcast to all consumers of that Subplan operator.

There are two types of subplans that Theseus encounters. The first is a *non-recursive subplan*. This is essentially like a remote procedure or function call. The reason for using this type of subplan is to leverage functionality that has already been written. In this sense, developers can build plan libraries and promote reusability, while maintaining simplicity in more complex plans. For example, a **GetPopularBooks** plan might call **GetPopularAmazonBooks** and **GetPopularBookpoolBooks** as subplans to leverage on data retrieval that has already been coded as a distinct plan.

The second type of subplan is the *recursive subplan*. These are plans that invoke themselves, building a nest of invocations until an exit condition is met and the recursion is unwound. Each invocation of recursion for such plans is known as a *recursive frame*. Recursive plans are characterized by the following properties: (a) they conditionally invoke themselves, (b) failure of the condition to invoke themselves leads to termination of the subplan (specifically, a Terminate operator). The HomeSeekers subplan is an example of this kind of subplan – it is called by a parent to process the first set of results and then calls itself repeatedly until it exhausts all of the house result pages.

It is also possible to encounter “indirect recursion”, where Plan A calls Plan B which calls Plan A and so on. This case is still classified as one of recursive subplan execution. However, the Theseus execution system ensures that only one instance of a plan per address space exists, letting the plan operators manage the nesting levels (recursions) themselves. **The general logic is:** at run-time, merge a single copy of all possible plans into a single address space. Thus, it should not be the case that recursive subplans are repeatedly merged.

Recursive and non-recursive subplans are implemented differently, since the former can potentially demand an infinite amount of parallelism, while the latter is bounded.

6.7.1. Transactions and Subplans

Each time a plan is called, it increments the stack frame for that plan (all plans initially start at 0). The stack frame ID is part of the transaction ID. Invoking a plan as a subplan is similar to invoking it as a standalone plan. Input and output adapters marshal data from the parent to the child plan. These adapters, like they do for the parent plan, augment the transaction ID. However, they know when they are being called by a parent plan, and instead of assigning a transaction ID, they append on a recursive frame ID. Thus, input and output adapters for subplans facilitate subtransaction management.

If transactions are voided at the subplan level, they only pertain to the subtransaction (and any other child transactions that subplan may have been responsible for). In other words, subplans cannot void the parent transaction. In addition to being intuitive, this is a useful security feature: a remote plan cannot adversely affect the data structures or execution of a parent plan. Also, it allows error handling to be fine grained – if a subplan has problems retrieving data from the network, it can restart just from its level.: there is no need to necessary restart the parent transaction(s).

6.7.2. Tail-Recursive Optimization

It may be possible to optimize recursive plans for the tail-recursion scenario. Basically, this is the case where a result arrived at by a recursive function can be expedited to the top level of recursion (assuming that the intermediate levels would normally do nothing except feed that data to their upper levels, and so on).

The optimization might work like this: the plan compiler would determine if the result a recursive call encounters is ever post-processed (i.e., is the result determination the last operator for a particular flow?) and, if not, it would send that result straight to the top level. Note that, since subplans are implemented via sub-transactions, the same physical resources (i.e., worker threads) used for the top level are used for every recursive level, so it would be possible to do this without a context switch.

The decision about whether and details about how tail recursion is optimized is a task left up to the implementation of the executor.

7. Termination

There are two aspects of termination that are addressed here, *plan termination* and *transaction termination*. The former can occur when a running plan will no longer be invoked. The second case happens when a transaction is guaranteed to have passed completely throughout the system. The second case is particularly difficult to discern because of the decentralized nature of dataflow execution in Theseus; termination is one of the only (if not the only) times when global synchronization of the system is required.

7.1. Plan Termination

Plan termination in Theseus occurs when (a) the caller has declared that no more input will be provided and (b) all transactions have been terminated. The issue of (b) is described further in the section below. For now, we focus on (a).

Callers (i.e., developers of Theseus clients) are partially in control of when plan termination controls. Based on the type of client being developed, plans may or may not terminate. For example, an interactive client typically reads a finite input file and sends the data contained within to the executor. Since the input file is finite, there will be some point at which all of the data has been passed to the system. It is at that point that the client denotes that “all input has been completed” (there is an API call for this). Keep in mind that the input file may not be well-formed and may only contain partial data for a transaction. For example, a simple plan that adds two numbers may only receive one of the numbers. In that case, the client may still declare that input has been completed, but will have to wait for the implicit (TTL-style) transaction termination process to occur before the plan terminates.

Some callers work on infinite input streams and thus the plan will never – normally – terminate. Consider a 24x7 web application that accesses Theseus via a Plan Server. In that case, the Plan Server receives network requests that contain plan input data. Since the application is 24x7, there is never any time at which input will end and, as long as the machine/server is available, so should the plan. Of course, there may be times when Plan Servers are cycled (for example, restarting JVMs because of bugs with the garbage collector or because of a new release of Theseus) – in that case, the Plan Server would denote that all input had been completed (just as in the interactive case).

7.2. Transaction Termination

Transactions terminate both per-operator and per-plan. Normally, when an operator processes input (including the final end-of stream, for all streams), that operator is done with that transaction, so that operator terminates any related data (such as state information) upon processing. A transaction is said to be terminated from a plan when all operators have terminated their transactions.

7.2.1. Explicit Termination

The most common way operators terminate their transactions is explicitly – by processing the final input (or the final EOS of the final input).

However, transactions can also be explicitly terminated if the Void operator is invoked during execution. In this case, the Void operator does not look at the value of the incoming data, just its transaction ID. It then coordinates the termination of that transaction from the rest of the system (other operators, input adapters, output adapters, etc).

7.2.2. Implicit Termination

As resources permit, operators will garbage-collect their queues and dispose of data belonging to stale transactions. These are transactions that have exceeded their time-to-live, as specified by metadata about the plan. The mechanism for garbage collection is directly related to how the Executor is implemented. See the “Implementation” chapter of this document for more detail.

Notice, however, that implicit termination by transaction TTL allows plans that experience unrecoverable errors to eventually terminate. Usually, the default error handling mechanism will attempt to “re-try” a plan. However, the plan writer may find it necessary to have some custom error handling flow. Still, this leaves it possible for execution to be executed indefinitely. To firmly guarantee termination in this case, when user-level error handling fails, the TTL approach to transaction can be useful.

The only drawback of implicit termination is that it may occur when the plan is actually still running – perhaps a particular network retrieval does take a long time. In this case, it is up to the system operator to correctly specify the TTL value in the plan metadata.

7.3. Detecting Transaction Termination at the Plan Level

It is relatively simple to support explicit and implicit termination at the operator level. However, it is more complex to deal with it at the plan level. As with Turing machines, you cannot prove that all dataflow graphs will halt. Obviously, since recursion is supported, infinite loops are possible – thus, the halting problem can be proven as unsolvable in the same way that it is for Turing machines (by contradiction). Furthermore, the fact that Theseus can be deployed indefinitely means that, theoretically, a plan could receive an infinite stream of data (no EOS). More alarmingly, since Theseus is extensible, there is nothing preventing an operator author from never emitting an EOS or from looping indefinitely.

In determining transaction termination at the plan level, we look at a restricted but very common subset of possible Theseus plans. These plans are called well-formed if they meet the semantic requirements of Theseus, one of which is:

given finite input, the plan is structured such that termination is possible

To be structured properly, the plan must obey the following two simple rules:

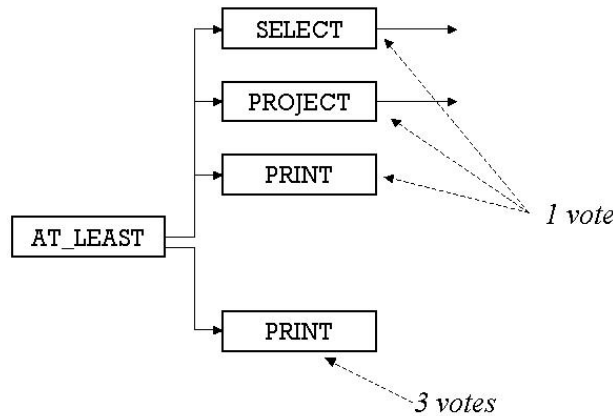
- (i) all conditional operators must exclusively use just one of their outputs per transaction*
- (ii) all non-conditional operators must use all of their outputs per transaction*

“Using all/one output(s) per transaction” can be explained best by example. Suppose a ChopChop operator exists that takes a TupleStream as input and produces three outputs: the first one outputs a stream of the first 10 input tuples, the second output contains the next 10 input tuples, and the last output contains all of the rest. First of all, notice that this operator cannot be declared conditional – there are certainly streams of more than 10 tuples, so output will not be exclusive. Now, suppose this operator receives an input stream of 9 input tuples. If ChopChop is written correctly for Theseus, it outputs just an EOS on the second and third output variables; if it does not, then the operator has not been written correctly.

It is important that operators be written correctly in order to guarantee transactional termination at the plan level. Although the requirement that conditional operators use only one of their outputs seems a bit restrictive, we believe that it does not preclude the development of any type of operator and it keeps things simple.

Given that a plan is well-formed, the executor using a “voting process” to determine transaction termination. Operators in the plan that represent the end of a branch in any flow are called *exit operators*. These operators are given k votes, depending on the parent flow(s) to which they belong.

For example, consider the following simple plan:



In this example, there are outputs of the `At_Least` operator, which is a conditional operator, so thus there are two flows. The lower flow consists of one branch and one operator (`Print`) and this operator is thus an exit operator. Meanwhile, the upper flow consists of three parallel branches that each have one operator (in each case, it is thus an exit operator). The plan must ensure transaction termination for either result of `At_Least`, so votes assigned to `Print` is 3, which is the sum of all votes given to exit operators of all parallel branches of the upper flow.

In the above example, the system deduces that a transaction has terminated from the plan when the system receives 3 votes. Since the voting process is done per operator, the termination process is decentralized, coordinated by a single global (synchronized) data structure. Thus, the voting process ensures correctness by counting and avoids race conditions by synchronization.

Operator developers and plan writers do not need to do anything in terms of the voting process. All vote determination and counting is handled automatically by the executor. However, to ensure correct transaction termination, plans and operators must be well formed in that:

- (i) *conditional operators must be declared so and obey the semantics described above*
- (ii) *plans should not be infinitely recursive*

Given that these two requirements are met, transaction termination at the plan level can be detected.

In the case of implicit (TTL) termination, the transaction garbage collector will coordinate the termination of the transaction at the plan level after it has cleaned the queues of all operators.

8. Error Handling

Theseus provides a default error handling mechanism and operators that can be used for additional, custom (plan-specific) error handling.

8.1. What Are Errors?

In Theseus, errors are defined as those instances where *the firing of an operator is halted by an unexpected situation that prevents further processing*.

Theseus distinguishes between errors that are temporary and others that are (seemingly) permanent, since the primary causes of run-time errors are operators that communicate with external/dynamic resources, thus facing traditional distributed system reliability issues. As described later, the Theseus default error handling routing addresses the former, more common case.

For example, take the case of the Retrieve operator. One of the potential hazards that may occur when using the Retrieve operator is that the remote web site being queried may be unavailable. This might mean that the site is either (a) temporarily unavailable (swamped with requests, intermediate network element failed/unavailable, timeout) (b) unavailable for longer/permanent period of time (site out of service, wide area network outage). In Theseus, case (a) is unexpected but it is recoverable – simply waiting and re-querying might be successful. However, case (b) is unexpected but un-recoverable. If the operator cannot obtain the data being sought, a data dependency will remain and the plan will not be able to terminate.

8.2. Default Error Handling

Theseus contains a default error handling mechanism that is designed to target the most common case: temporary errors. The general approach used is to simply void the transaction that encountered the error and attempt the same transaction again. Since there is automatic parallelism inherent in its dataflow environment, Theseus must ensure the synchronization associated with halting and then re-starting plan execution.

To accomplish this, Theseus contains an operator-level mechanism to report errors and a plan-level mechanism to synchronize upon error reporting and restart the plan. In the case of the former, the system detects exceptions and failures during operator processing (more about this below). Upon such situations, the operator generates an enablement called “_error”. Generation of this enablement triggers the plan-level mechanism for synchronized voiding and restarting. However, to support this plan-level mechanism, all plans need to be augmented during the compilation phase. Specifically, the dataflow graph for a given plan is modified in the following way:

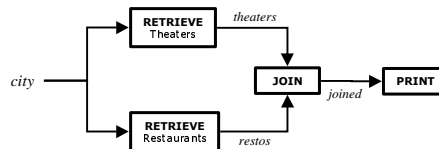
1. All plan inputs $i_1..i_n$ are identified
2. A Restart operator is inserted into the plan. The input to this operator are $i_1..i_n$ plus an additional `_voidOnErrorComplete` enablement.
3. A Void operator is inserted into the plan. Its input is an `_error` argument.

Thus, if any plan operator produces the `_error` enablement, the Void operator fires. This operator voids the transaction – it knows what transaction to void because that is encoded in the tag for the `_error` enablement it receives. The Void ensures that all operators have voided data corresponding to that particular transaction before producing its `_voidOnErrorComplete` enablement. This is, in turn, consumed by the Restart operator, which has also previously

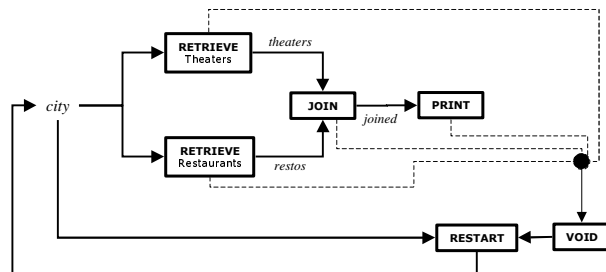
received all of the plan input arguments. The Restart operator than re-broadcasts the $i_1..i_n$ input arguments. Thus, the effect is the same as when the plan had initially started: a new transaction is started, consumers receive the input data, and the Void operator is once again prepared to deal with the potential generation of the `_error` enablement.

Below is a visual example of this default error processing routine:

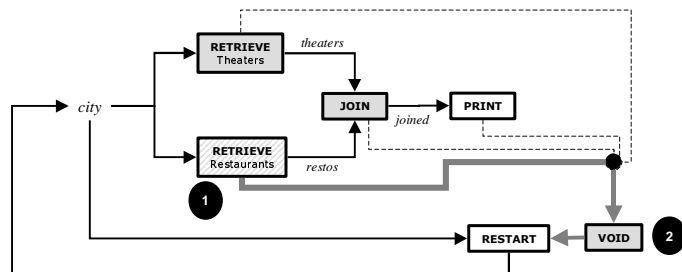
The original plan: print out the joined retrieval of restaurants and theaters.



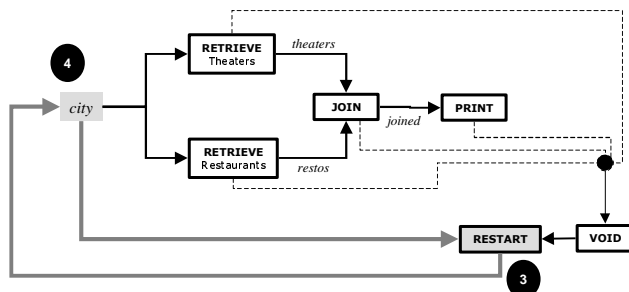
The augmented plan: the Void and Restart operators have been added *during compilation*. Notice that every operator (except Restart) can output an error enablement (shown with dashed lines) to Void.



Error in action: The filled boxes indicate active operators. The lower retrieve (1) is shown encountering an error. This causes an error enablement to be sent to the Void operator (2).



Void and restart: After all operators have been halted (for that transaction), Restart is signaled (3). This causes the regeneration of the City parameter (also fed back to Restart), effectively restarting the transaction.



Note that it seems a race condition exists: multiple operators could produce the `_error` enablement at the same time. However, this is actually a non-issue because the semantics of Void are to remove all data for a given transaction at every operator (except itself). Thus, the

duplicate `_error` enablement (probably residing in the queue of the Void operator) is also removed. Furthermore, if that enablement were somehow in transit (suppose operator I/O model involving a network), the TTL transaction voiding aspect of Theseus would eventually remove that extra `_error` enablement.

The Restart operator maintains internal state and counts the number of restarts. If this count exceeds the system specified default, the input parameters are not reproduced, and thus no new transaction is started.

If desired, the default Theseus error handling mechanism can be turned off. This is specified at the time the plan is run. In such a case, none of the plan operator will produce an `_error` enablement – thus, the Void and Restart will never occur.

8.3. Plan-Specific Error Handling

Plan writers can design their own error handling routine with the Void and Restart operators. Although they cannot intercept the production of the “`_error`” or “`_voidOnErrorComplete`” enablements, they can use the `ON_ERROR` feature of the Theseus plan language to consume a custom error-signaling enablement and then Void and Restart as necessary.

9. Instrumentation and Debugging

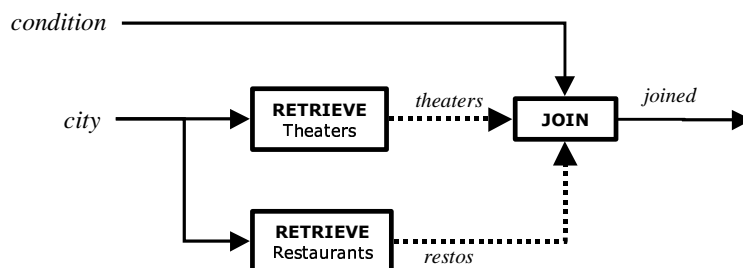
Instrumentation and debugging is particularly challenging in a dataflow environment. The main problem is that execution is decentralized: halting the execution of the system for purposes of, say, debugging is a synchronization demand that atypical of what normally occurs during dataflow execution.

To support both instrumentation and debugging, Theseus V2 supports a general event notification infrastructure to the executor. The general idea is to have components or modules consume the events that occur in the executor, either synchronously or asynchronously. For example, a debugger would require synchronous communication of events (execution should be halted until directed by debugger) whereas a visualization engine might not care necessarily about being notified exactly when an event happened.

9.1. Events

Although many possible events could be generated by the executor, Theseus V2 will initially focus on two events that appear to be the most useful/profitable: (a) the arrival of data at a queue and (b) the firing of an operator.

The first type of event is generated by the executor is that *when an item of named data is consumed* (abbreviated as “data event”). For example, in the figure below, this the consumption of either “theaters”, “restos”, or “condition” by the Join operator shown is considered an event. An *item* of data refers to a tuple or EOS in the case of streamed inputs and to each instance of a non-streamed input (static data excluded). Note that the consumption of data by the consuming operator does not necessarily imply that this operator will immediately execute – it simply occurs when data arrives, allowing data in the system to be tracked at a fine level of granularity.



Note: for this figure, streamed inputs are denoted with dashed lines

The second type of event is generated by the executor is *that when an operator firing rule has been met* (abbreviated as “operator event”). Thus, the goal is to capture an operator before it actually begins an execution cycle. For example, in the figure above, the existence of tuples on the “restos” or “theaters” queues of Join, along with the presence of “condition” is an instance of the operator firing rule. More generally, this event is generated depending on the operator firing rule, so it is specific to every operator. Thus, this event type allows for the tracking of operators, as opposed to the tracking of the presence of individual tuples/non-streamed elements that a data event allows.

Both types of events can be processed either synchronously or asynchronously, as described below.

9.2. Callbacks

A *callback* is a function that is invoked whenever an event occurs. Callbacks are *registered* for a particular event. For example, if one wanted to write a debugger, he would register callbacks with the executor so that he could track the existence of data and the execution of operators as they occur.

Callbacks are synchronous. Execution is halted until all callbacks have returned. Any number of callbacks can be registered to each of the two events described in the previous subsection.

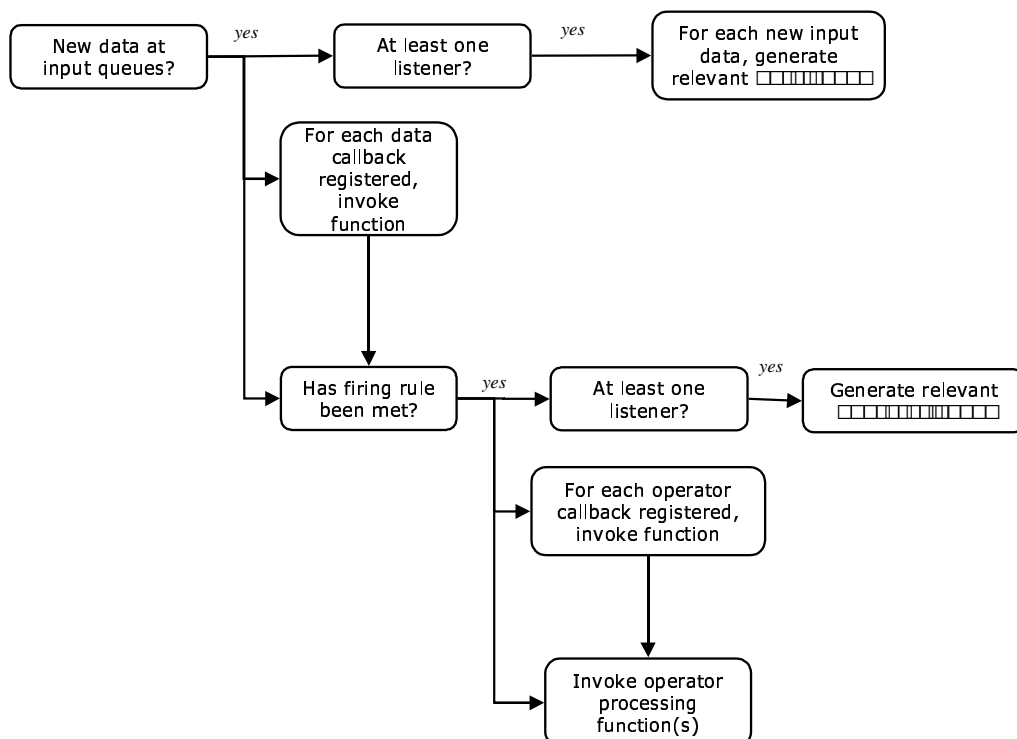
9.3. Event Queues & Listeners

Event queues are similar to callbacks except that they are asynchronous. Whenever data arrives at a queue or the firing rule for a particular operator is triggered, the system generates pushes events into specific event queues. Listeners are used to read/process this queue.

Similar to callbacks, clients register listeners. Clients are responsible for processing their own queues. There is no guarantee of order on the events generated. From the perspective of the system, listeners are easier to process than callbacks, since they do not significantly affect execution and because the system does not need to know about every asynchronous listener.

9.4. Generating Events and Callbacks

It is useful to describe the general internal algorithm for event processing in the context of debugging. Assume that an operator knows that some set of input data has arrived at one or more queues (whether it polls or is notified about this situation is irrelevant). Given that assumption, operators consume data as follows:



10. Optimizations

There are several local and global optimizations possible in Theseus, both static and dynamic in nature. While it is not the purpose of this document to detail future optimizations, here is a brief list of ideas being considered. These ideas are either active areas of research or have been raised during meetings.

- **Speculative execution:** executing plans (or parts of a plan) in advance, based on the likelihood that such a request will be made anyway. Executing in advance might make better use of available CPU cycles, since many Theseus plans tend to be I/O bound in nature.
- **Serial Flow Compression:** Related to the dataflow concept of macro operators, the idea is to not waste the messaging overhead when executing serial flows. It is especially wasteful to set up serial pipelines if the data between the operators is not streamed and thus cannot be processed in parallel.
- **Smart (Eductive) Transaction Voiding:** During execution, it is possible to detect cases where the failure of one of several parallel dataflow negates the need for the others. In these cases, like the education model, we would like to void the work being performed.
- **Transaction Bundling:** Handling and merging multiple inputs (even from different transactions) may be useful in cases where the input is often repeated and the cost of acting per input is high. Instead, the inputs could be merged and the processing results shared. A prime candidate for this type of optimization is the Retrieve operator.
- **Dynamic Application Servers:** Through plan execution profiling, we could detect heavily trafficked parts of a plan, or just those parts that are bottlenecks. For these plan fragments, we could automatically convert to a subplan that, for example, might have more worker threads than would be possible when that plan fragment was in the original plan (competing for resources).

Implementation

The document thus far has focused on the design of the system. We now turn to implementation, specifically covering those aspects of the implementation which are the most contentious, confusing, and/or most important.

11. Implementing the Executor

11.1. Aspirations

To support the parallelism demands of streaming and transactions, Theseus needs to consist of a highly decentralized, asynchronous infrastructure. Some of the general goals we have kept in mind while designing the system include:

- DO avoid centralized data structures (unless they are read-only)
- DO reduce the amount of data copying required during execution
- DO allow plan authors, system administrators, and the internal system itself convenient mechanisms improve its own capability for scalability/parallelism
- DO realize that different operators have different scaling needs (i.e., Retrieve vs. Notify), based on their implementation semantics and CPU usage patterns
- DO NOT require operator I/O to be synchronous
- DO NOT allow the parallelism to get out of control - some mechanism, like k-triggers [Arvind et. al. 1981] is needed
- DO NOT spend more time context switching than processing
- DO NOT allow all system resources (i.e., threads) to be exhausted so that the system effectively deadlocks (as can happen with uncontrolled dynamic dataflow systems [Ianucci 1988]).

11.2. Theoretical Parallelism vs. Real Parallelism

The above phrase *capability for scalability/parallelism* is used because, although Theseus is designed as a virtual dataflow system that can manage many parallel activities, the real parallelism possible during execution ultimately depends on the number of CPUs used. Theseus assumes that - either through multiple-processor machines, SMP, or cluster technologies - the underlying OS/hardware may be able to support high degrees of parallelism.

However, Theseus does not (a) attempt to optimize itself to work with any one particular scheme for attaining real parallelism or (b) intend to discourage users of single CPU machines in any way. Rather, Theseus is built with enough hooks so that system operators can tune the number of concurrent transactions they want to support, as well as the scaling behavior of their plans and operators.

11.3. Executor Implementations

Theseus is design modularly, so that components such as the executor can be easily replaced, as long as the abstract API remains constant and some basic abstract ideas of execution (asynchronous, streaming, etc) are implemented. These ideas are described in more detail in the Execution section.

In this section, we describe current models of execution that we are considering for implementation, focusing on the two that are currently the most popular: one that provides basic functionality and one that aspires for higher performance.

11.3.1. The Basic Threaded Implementation

There are two goals being sought in the *basic threaded implementation* of the Theseus executor. One is simplicity: a basic implementation will be easier and quicker to develop and allow us to reach the system integration stage quickly. The second is to act as a data point: the basic implementation is not as ambitious as the manager/worker style (described later) and we can thus compare the two to accurately measure the practical effectiveness of the manager/worker style.

The highlights of the basic implementation are:

- each operator instance is associated with one thread (to prevent deadlocking)
- operator instances keep track of their own transactions and metadata
- producer operators explicitly notify consumer operators when they have produced data to a consumer queue

An example of the basic implementation is shown below. In this example, a join is performed on two relations that are retrieved. The first figure shows the two Retrieve operators and a Join operator. The Retrieve operators consume *queries* and produce *data*. The Join operator consumes *left* and *right* relations, along with a join condition, and produces the *joined result* (unlabeled in the figures – not important). Notice that, in the first figure, each Retrieve has 3 queries on its queue and the Join has no data on any of its left and right queues (however, the static condition does exist). At this point, the Retrieves meet the firing rule criteria, but the Join does not. In the second figure, we see a work-in-progress. One of the Retrieves is working faster than the other (different query queue sizes) and the Join is attending to queues on demand.

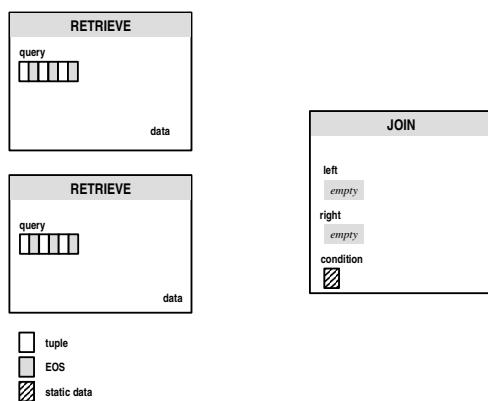


Figure 11.1a: Basic model – before Join

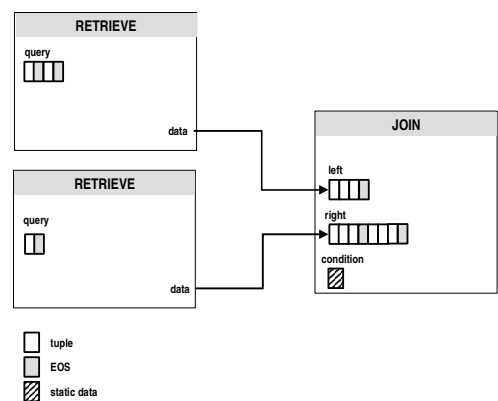
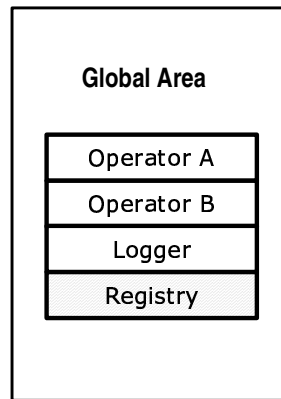


Figure 11.1b: Basic model – during Join

The threading model of this approach for a particular plan can be summarized in the figure below. Each of the boxes represents a thread running in the address space of the process (plan).



In this figure, notice that the Registry is included as one of the threads. As described earlier (see Invocation), if the external Registry service is not available, a thread is allocated to it in the Theseus runtime.

11.3.2. The Manager/Workers Implementation

The basic manager/worker runtime system consists of *manager threads* and *worker threads*. The former manage the execution of a particular operator instance while the latter are used purely as computational resources. The manager/worker model is a solution that attempts to decentralize operator execution as much as possible, retaining only enough centralization to handle critical synchronization and state management responsibilities.

A manager thread is assigned to every instance of an operator in a given plan. For example, if a plan has three Select operators, there are three Select manager threads. Each manager interfaces with 1..N worker threads, which are used purely for computation. The number of worker threads is scaled based on (a) the blocking nature of the operator, (b) the number of concurrent transactions the system operator intends to support per plan instance, and (c) whether or not that operator is part of a recursive subplan (see below). Details on scaling based on these factors is provided later in this section. Note that all operator instances have at least one manager and one worker associated with them.

To understand how execution works under this model, consider the simple Reuters-News Fetch plan below. In this example, an initial Retrieve is done to gather the list of links to news stories. This relation of URLs is reformatted as queries and pipelined to a second Retrieve operator. Finally, this information is cataloged in a database.

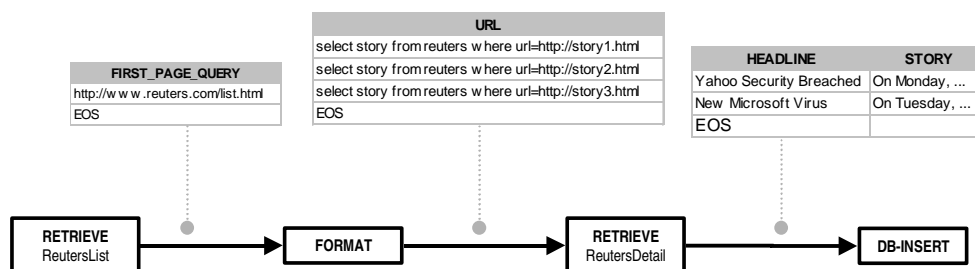


Figure 11.2: Reuters-News Fetch

Next, we show how the manager/worker model could be tune the deployment of this plan to achieve minimal functionality (Figure 11.3a), decent performance for one transaction (Figure 11.3b), and decent performance for multiple transactions (Figure 11.3c). In the first case, shown by Figure 11.3a, we have at least one manager and worker. While the plan will function correctly, there will likely be a bottleneck at the second Retrieve, which is executing a query for each tuple it receives. If it only has one worker thread, all of the pipelining is not really useful for prior parts of the plan – every tuple still has to wait in line for access to this second Retrieve.

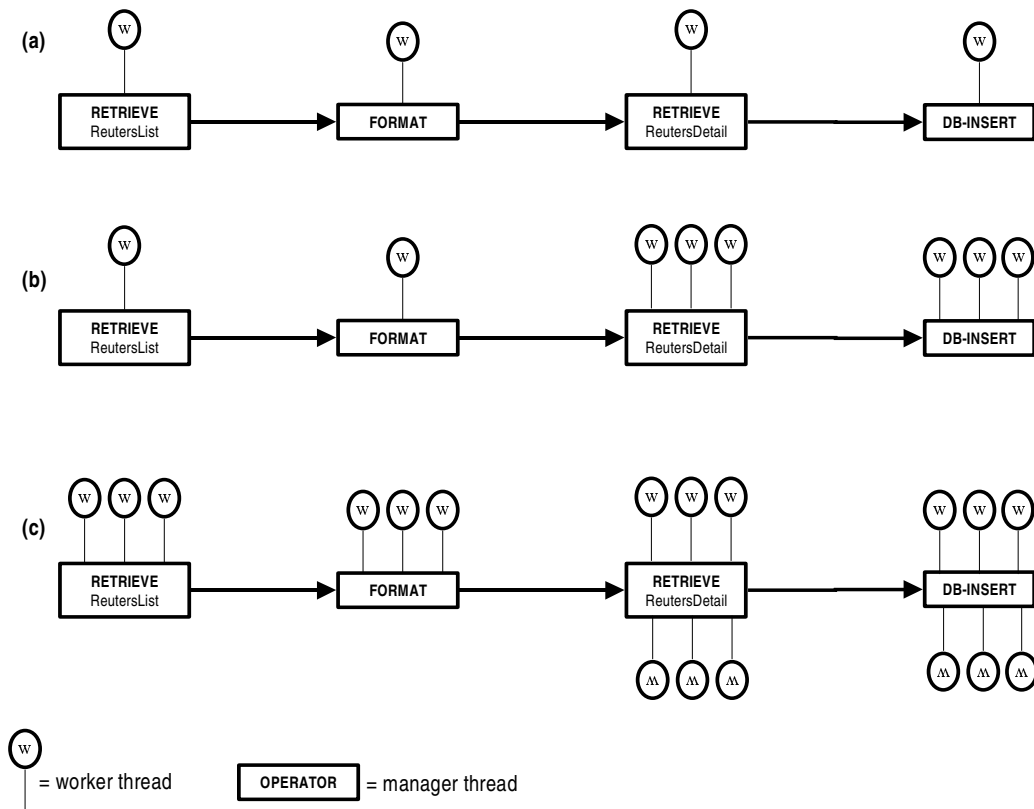


Figure 11.3: Reuter-News fetch runtime models for (a) single transaction, basic execution, (b) single transaction, better performance, (c) multiple transactions, decent performance

In the second case, illustrated in Figure 11.3b, there are multiple worker threads for the second Retrieve and for the Db-Insert operator. This allows a single transaction to theoretically run w times faster at a given operator instance, where w is the number of worker threads used by that instance. This type of design is particularly useful if the source accessed by the second Retrieve is slow – a system with only one CPU can still achieve good speedup by making those requests in parallel. Also note that, in this design, concurrent transactions are not handled well – for many operators, there is still only one worker thread.

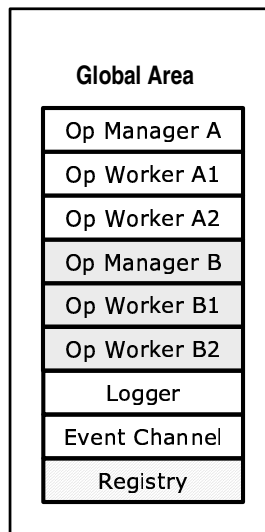
Finally, in the third case (Figure 11.3c), we show a deployment that can handle three concurrent transactions. Notice that our algorithm for scaling the system for multiple transactions is something like:

- create at least one worker thread at each operator instance per concurrent transaction support desired

- create some optimal pool p of worker threads for I/O-bound operator in the presence of concurrent transactions

There has been research into developing a cost model for this type of parallelism [Motwani et. al. 1998], but automatically achieving that optimality is not discussed here. Instead, we assume that, based on web site and plan profiling, an optimal arrangement of worker threads can be determined.

The threading model of this style of implementation can be summarized below:



11.3.2.1. Resource Allocation Issues (Transaction Concurrency vs. Parallelism)

It is important to note that, simply by increasing the pool size of worker threads to handle multiple transactions does not necessary allocate worker threads evenly among transactions. For example, in Figure 11.3c, suppose three simultaneous transactions occur. Although they occur simultaneously, they will experience different latencies as a result of the first Retrieve and arrive at the Format and second Retrieve in some linear order that differs by milliseconds. Suppose, in this case, that the first one reaches the second Retrieve first, with a table of six stories to retrieve. Since each news story will be retrieved with a separate thread, the resources of the second Retrieve will be completely dedicated to executing the first transaction only, despite the arrival of the other two transactions just milliseconds later!

Thus, there are cases where reserving resources for high transaction concurrency would seem important. However, there is also the desire to allow a single transaction to occupy all of the resources it can get. For example, if there were only one transaction (as described above), and not three, we would like that transaction to actually be able to use all of the worker thread resources at each of the operators.

Flexible resource allocation algorithms and designs have been studied extensively in other areas of computer science and are not a focus of the research on this project. Instead, we will implement a simple mechanism for allowing resources to be allocated fairly, based on the two types of allocation scenarios described. Our mechanism, called *work-stealing*, is described later in this section. However, before describing the details of the work-stealing algorithm, we need to describe exactly how I/O will be done between operators.

11.3.2.2. Operator I/O Details

In the manager-worker model of execution, data is passed between manager queues but is operated on by worker threads. Worker threads are threads spawned by the manager thread, so they have access to the same address space as the manager thread and can thus easily access its incoming work queues. However, since there may be several worker threads, access to these common queues must be controlled.

Two approaches are discussed here. It is likely that the first will be implemented, since it allows for the most flexibility and retains the ability to scale for parallelism or concurrency, as desired. The second approach is included because, if given enough thought, it might allow greater asynchronous computation. However, there are some problems with the second design and its implementation might be more complicated. Until these problems are worked out, it remains an inferior approach to the first method.

Before describing these methods in detail, we first make a note about the way workers monitor and/or are notified about data on their queue.

Combining Event-Driven and Polling Approaches

There are usually two choices a designer makes when implementing some sort of messaging system: will it poll for messages or will it be notified about them in some type of event-driven manner. Both have advantages and disadvantages. For polling, the main obvious problem is that a lot of polling will continue to be done when it is not relevant (no data in the system or data has already been processed). This taxes resources heavily and is generally considered to be an inferior approach. However, one strength of polling is that it works well when lots of data accumulates a queue. In this sense, each poll is profitable. However, the unfortunate fact is that long term polling is very inefficient.

In contrast, an event driven system only monitors its queues when notified. This is a demand driven approach to queue processing. In this case, for Theseus, a producer operator output queue would notify a consumer operator input queue about the existence of data. In general, this is a nice approach – the consumer only worries about processing data when data is actually on the queue. However, in a streaming environment, this creates a lot of messages. Also, these messages are redundant: if we are in the middle of processing a stream, we are pretty sure that there will be more tuples – we do not need to be told about this.

In Theseus V2, we seek to combine event-driven and polling approaches. The general behavior could be as such: when generating the first tuple of a stream, the producer operator notifies the consumer of an event (meaning “start monitoring your queue”). The consumer then starts polling – no more event messages are required. This polling continues until an EOS is reached. Then, the queue returns to a “wait state” until the next stream (event) occurs.

Ring-Based Queue Processing

In this model, queues can be thought of as a specialized form of stacks. As with stacks in general, data can be PUSH-ed onto them and POP-ed from them. However, the stacks are also specialized in that they will support the ability to POP the next data for a specified transaction. It is this special operation that will enable fair concurrency policies to be maintained.

At runtime, producer worker threads process data and write their results directly into the queues maintained by the manager of each consumer operator. They essentially PUSH their results onto the stack as necessary. Consumer threads POP data from this stack, but they have to be careful about synchronization. Since there may be many worker threads, it is undesirable to have all of them trying to access the shared stack at the same time. To solve this problem cleanly with

minimal overhead, we can enforce serialization of queue access using a standard ring-based access model [ref?]. This is a common, well-known solution for reducing contention for a shared resource – the token ring model of networking is based upon this same approach.

In the ring-based approach, worker threads do not try to POP the shared queue until they have what is known as a baton. This data structure contains some metadata, but most importantly, it empowers the worker to access the shared queue. Thus, management of access to a common queue is decentralized. Once data has been POP-ed from the queue, the worker thread passes the baton onto the next worker and then processes the data it has acquired. Thus, the next worker can access the queue while the previous worker is busy computing.

Queue handoff is synchronized by the manager, so that worker threads can run in parallel. Workers are organized so that each only points to one neighbor, and that the last worker points to the first worker, forming a complete circle. This is shown in more detail below:

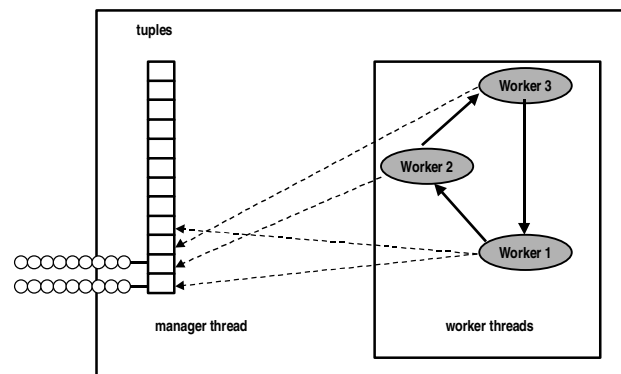


Figure 11.4: *Ring-based queue access*

In the figure, notice how Worker 1 accesses the tuples queue of the operator and then passes off the baton to Worker 2, and so on. This guarantees collision-free access of the queue with the minimum amount of synchronization overhead. There is the cost of serializing access to the queue, but this can only be overcome by using queue offsets (described later), which – for other reasons - is a more costly solution. Furthermore, the serialized access in this ring-based approach incurs minimum penalty, since the workers handoff the baton *before* processing the data they have acquired.

For a more complete example of this type of processing, consider a simple combination of Retrieve and Select, shown in Figure 11.5 Deployed using a ring-based approach, the Select resembles the operator shown in Figure 11.4, as does the Retrieve operator.

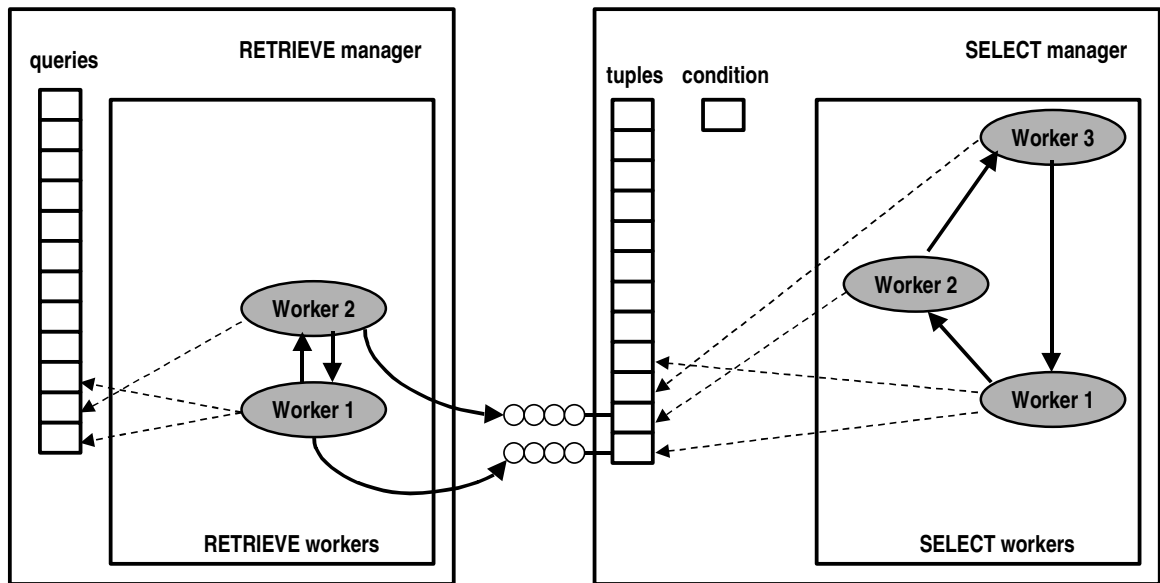


Figure 11.5: Details of ring-based queue processing in the Retrieve-Select example

A modified version of the ring-based approach could be implemented where there were subsets of worker pool, assigned to different transactions. This would allow high concurrency without the threat of resource-hogging during a highly parallel transaction. When only one transaction is active, some form of work-stealing (described below) could be applied.

The Queue-Offset Method (*Alternative Approach*)

To enable high levels of concurrent access without incurring any read/write hazards, worker threads will access common queues at individual offsets. Similar to hash-based routing, reading a common queue at a unique offset ensures all incoming queue objects will be handled and that they will be handled by distinct computational resources.

There is also one caveat to the behavior of the worker threads. They are assumed to process when the firing rule is activated. The firing rule is not active until all of the non-streaming inputs for a particular transaction arrive, so worker threads will not be woken up to start monitoring their queue offsets until the manager thread has received all of the non-streaming inputs for that transaction. Thus, worker threads will only monitor streaming queues – non-streamed queues will be managed by the manager thread. Since the flow rate on non-streamed queues is much smaller than that for streaming queues, a single manager thread can likely handle the anticipated demand. Also, non-streamed queues are more likely to have static data associated with them – data that does not change and applies to every transaction.

The easiest way to understand basic operator I/O is by way of example. In our example, suppose that a Retrieve is followed by a Select, as shown in Figure 11.6. Suppose that we would like to support two concurrent transactions and that we would like some degree of parallelism in processing the Select (assume that our streaming rate between Retrieve and Select is such that 10 tuples at a time are streamed). Thus, our deployment could be something like Figure 11.6, where there are two worker threads for Retrieve (one for each concurrent transaction desired) and two pools of worker threads for Select (one pool for each transaction).

In this deployment, we view the sharing of resources for concurrent transactions as being an optimal allocation strategy. Thus, the grey-filled worker threads in Figure 5.6 correspond to one transaction while the unfilled worker threads correspond to the other transaction. Thus, if both

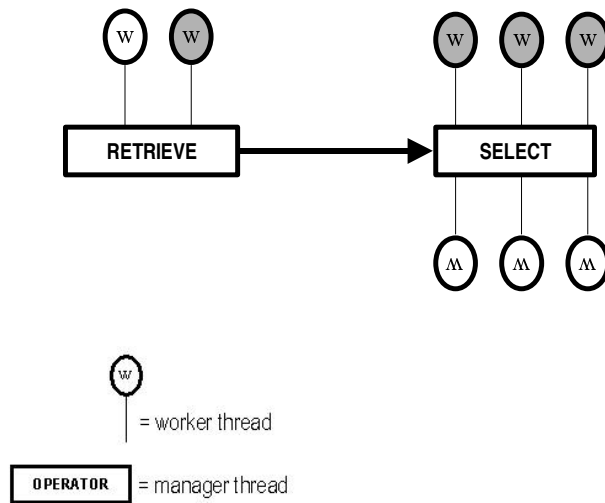


Figure 11.6: Retrieve-Select example

transactions occur at once, each transaction will be able to use one Retrieve thread and three Select threads.

The details of how data is passed between Retrieve and Select is shown in Figure 12.7.

Each worker in this figure is labeled corresponding to the operator, transaction, and stream instance it to which it is applied. For example, Worker SB2 means: applied to Select, transaction B (remember, we want to handle two simultaneous transactions, A and B), stream instance 2. Here, “stream instance” roughly corresponds to “pool member”. For example, Select maintains pools of 3 workers per transaction.

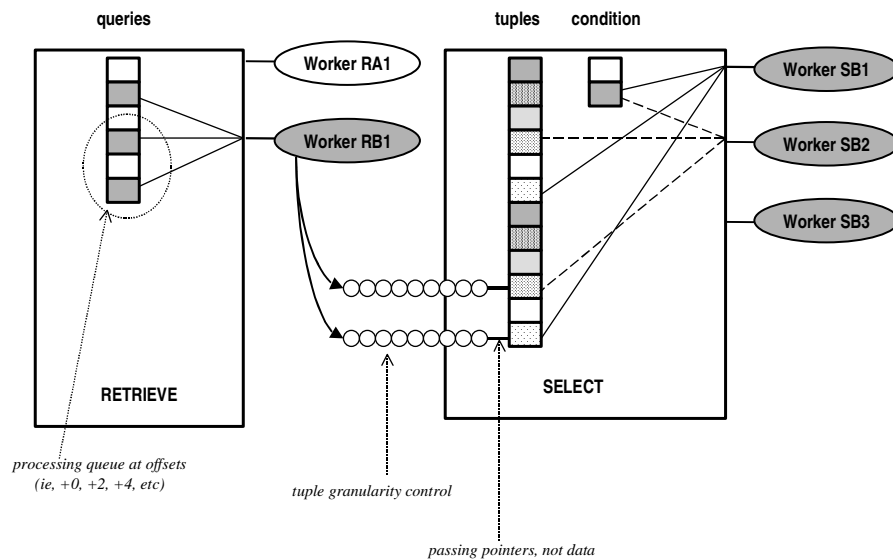


Figure 12.7: Alternative operator I/O scheme using queue offsets

Key features shown in Figure 12.7 are:

- worker threads process data at offsets: Worker RB1 only reads at offsets of +0, +2, etc. Similarly, multiple workers for a common transaction may share non-streamed data (the condition queue of Select), but work on independent batches of streamed data (the tuples queue of Select), such as is true with Workers SB1 and SB2
- pointers to data are passed, not the data itself: reduces data copying, allows flexible control of streaming granularity

Another feature **NOT** shown in the above figure is the use of *circular buffers*. Using these types of buffers allows the workers to constantly poll (they never reach the “end” of a list) and also allows the queuing to be bounded in some fashion. Optimizing the allocation of space in these circular buffers will be detailed in a future revision of this document. For now, worker threads will know (apriori) the size of the manager buffer and will be able to enforce the circular access pattern themselves.

11.3.2.3. Dedicated Worker Pools vs. Global Worker Pool

It might occur to you that another way to deploy the manager-worker execution model is such that there remain distinct manager threads, but that there is some global pool of workers that can compute as necessary. Thus, worker threads are assigned to work on data as it becomes available in queues, loading whatever operator template is necessary.

This is very similar to how real dataflow systems work, but there are a few reasons why this is probably not a better architecture:

- **Theseus is a virtual dataflow system:** Since it is implemented in software, the cost of reassigning worker threads to arbitrary tasks has a higher cost than if the system was deployed as hardware.
- **Higher overhead for possibly very little gain:** Even if there were no costs to deploying a global pool approach, it’s not clear that this has a substantial performance impact. For one, while it does cost something to create/destroy threads, it doesn’t really cost that much to maintain them. So, having a bunch of worker threads at each operator isn’t necessarily a waste. Second, using all of the worker threads to attack a problem in parallel really only makes sense for I/O-bound operators like Retrieve. In all other cases, you likely will not have the real CPU availabilities to meet the demand.

11.3.2.4. Work Stealing

Two of the design philosophies of Theseus V2 are: high concurrency & high parallelism. However, for the two scenarios described earlier, when both high concurrency and parallelism are demanded, how should resources be allocated?

This will be detailed in a future version of this document, but the basic idea is to allow unemployed worker threads to share the work of an employed worker by monitoring selected parts of the employed workers queue. Essentially, this means that worker thread Y will sometimes read the manager input queue at a different offset than usual, say the offset used by worker thread X. Of course, the extra cost here is the synchronization between X and Y, but since this will only occur in cases where one of the two workers has some free time, it is theorized that having more workers attack a common problem overcomes any penalties synchronization might incur.

A case where work stealing may be very useful is when the system is executing multiple transactions that vary widely in the data they generate. Consider three transactions related to

searching for books that are invoked simultaneously: one is searching for all books written by “John A. Smith”, another for books written by “Jane Z. Smith”, and a third for books written by “Smith OR Thomas OR Anderson OR Jones”. It is likely that the first two transactions will result in less data than the third. In this case, the worker threads for the first two transactions will finish early, with nothing to do, while the third is busily processing its volumes of tuples. Work stealing would allow the workers for the first two transactions to “snoop” on the queue(s) of the third transaction, helping out if appropriate.

The idea of work stealing has been used in other asynchronous messaging systems, such as those described by [Lea 1999] and those described in the Telegraph project at UC Berkeley [Culler et. al. 1998].

12. Plan Operators

The following is a list of operators that are provided in conjunction with the executor. Although developers can write their own operators, many useful plans (including all of those referred to in the examples of this spec) can be written using the default set.

- Stream Operators

- o `apply (Function fun, Stream in_stream :)`

`apply` applies `fun` to `in_stream`

`fun` should have type `(Object -> unit)`. `fun` will be applied to each value that is part of `in_stream`.

`apply` allows a plan writer to just write a function to do what they want with each part of the stream, and not worry about writing a full blown operator.

- o `atleastone (Stream in_stream : Stream out_true, Stream out_false)`

Outputs `in_stream` to `out_true` if it contains at least one value, otherwise to `out_false`

`atleastone` allows plans to differentiate between no stream and an empty stream. These are subtly different concepts. No stream will not be passed on to other operators/subplans, but an empty stream will be. This operator works as a halt to pointless [i.e. infinite] recursion. No output occurs if `in_stream` has no data, but if it does contain data each value in `in_stream` is output as soon as it is seen to `out_true`

- o `difference (Stream universe, Stream sample : Stream negation)`

Computes the set difference between `universe` and `sample`.

Due to the nature of set difference, `difference` cannot output anything until all of `sample` has been seen and an EOS has been received. Once that occurs `difference` will begin outputting all values from `universe` that did not appear in `sample`. This will occur even if EOS has not yet occurred on `universe`.

It might be useful to provide a vararg set difference, although this would be nothing more than a union and set difference combined operator.

- o `filter (Function fun, Stream in_stream : Stream out_stream)`

`filter` uses `fun` to select which parts of `in_stream` to output to `out_stream`.

`fun` should have type `Object -> Boolean`. `fun` will be called on each value in `in_stream`. If `fun` returns `true` then that value will be made part of `out_stream`. When EOS is seen on `in_stream` it will be made part of `out_stream`.

- o `fold (Function fun, Stream in_stream, Object initial_value : Object result)`

Applies `fun` to `in_stream` returning a result.

`fold` applies `fun` to each value of `in_stream`. `fun` should be of type `(Object * Object) -> Object`. The first argument is a part of `in_stream`, and the second argument is either `initial_value` if this is the first time `fold` has run, or the result of the previous run. The return value of `fun` in the most recent invocation when EOS occurs on `in_stream` is made the value of `result`.

`fold` is inherently serial. It might be useful to define `folda`, or `fold` associative, which by name would communicate that `fun` was associative. `folda` would likely need another argument `id` for the identity element to be passed as `initial_value` in parallel invocations.

- o `headtail (Stream in_stream : Object head, Stream out_stream)`

Extracts one element from `in_stream` and puts it in `head`. The rest of the stream is output on `tail`.

`headtail` allows operation on a `Stream` value by value. This is useful for interaction with legacy operators which do not support streaming arguments. `head` will be a value that is part of `in_stream`, and `out_stream` will consist of the rest of the stream. Note that due to the absence of guarantees on ordering in streams, `headtail` is really `valuerest` since there is no notion of 'first' a stream.

Output will occur as soon as there is sufficient data. `head` will be output when the first value of `in_stream` is seen, and `out_stream` will commence output when the second value is seen. In the case when `in_stream` is an empty stream, `head` will not be output, and `out_stream` will be an empty stream.

- o `makestream (Function fun : Stream out_stream)`

Constructs `out_stream` by repeatedly calling `fun` until it returns `null`.

`fun` should have type `unit -> Object`. `fun` will be called from repeated invocations of `makestream` until it returns the sentinel value `null`. Each nonnull result that `fun` produces will be put to `out_stream` as soon as possible. When `null` is produced, EOS will be put to `out_stream`.

- o `map (Function fun, Stream in_stream : Stream out_stream)`

Applies `fun` to each piece of `in_stream` and returns the Stream of results.

`fun` should have type `Object -> Object`. For each value in `in_stream`, `fun` will be called with that value. The result of the call to `fun` will be made part of `out_stream`. If `fun` returns `null` that result will be ignored. Whenever EOS is seen on `in_stream` it is put to `out_stream`.

`map` allows a plan writer to just write a function to do what they want with each part of the stream, and not worry about writing a full blown operator.

- o `merge ((Stream in_stream)* : Stream out_stream)`

Combines the `in_stream`'s into `out_stream`. Duplicates are allowed.

`merge` takes a variable number of input streams, and gloms them together into `out_stream`. `merge` does not care about typesafety. `merge` is useful when the stream is not a tuplestream. In the latter case, `union` should be preferred.

Output begins when the first value arrives from any of the `instream`'s, and ends with EOS only after EOS has been seen on all the `in_stream`'s.

- o `partition (Function fun, Stream in_stream : (Stream out_stream)*)`

Uses `fun` to divide `in_stream` into the `out_stream`'s.

`fun` should have type `Object -> Integer`. `partition` will call `fun` with each value that is part of `in_stream` and use the return result for an index into the variable length `out_stream`. The index will denote which `out_stream` to put the value to. Valid indices

will be consecutive integers starting at 1. Thus if there are two `out_stream`'s, then the valid indices will be 1 and 2. Invalid indices will cause the value to be ignored, and not output to any of the `out_stream`'s. EOS will be output to all `out_stream`'s once EOS is seen on `in_stream`.

- Relational Operators

- o `accumulate (TupleStream in_stream : Relation out_relation)`

Takes `in_stream` and converts it to `out_relation`.

Each tuple of `in_stream` is made part of `out_relation`. Order is arbitrary, but then again a Stream does not contain any notion of order. The Relation is output only after the operator has seen EOS on `in_stream`.

- o `aggregate (String column, String func_name, TupleStream in_stream : Object result)`

Takes `in_stream` and applies the `func_name` function to the specified `column` and returns the `result`.

`func_name` should be one of "sum", "min", "max", ... `result` is output with the result of this computation.

Note: it might be good to provide a generalized form of `aggregate` that essentially does a `fold` on a certain column.

- o `join (String cond, TupleStream left_stream, TupleStream right_stream : TupleStream result)`

Joins `left_stream` and `right_stream` according to the `cond`.

`cond` will be a sequence of join conditions connected by "AND". Each join condition will consist of "LATTR ? RATTR" where LATTR is an attribute from `left_stream`, RATTR is an attribute from `right_stream`, and ? is a comparison. All 6 comparisons are supported.

`join` will start producing output once tuples have been identified that are definitely part of `result`. EOS will be produced only after the join is done. This requires that EOS has been seen on both inputs.

Note: it might be beneficial to provide a varargs type join, where arbitrary number of tuplestream's are joined.

- o `namerelation (String name, Relation in_rel : Relation out_rel)`

Names the `in_rel`. This is useful just before operator `exportrelation`.

`namerelation` changes the name of `in_rel` so that it has the name, `name`.

- o `project (String cond, TupleStream in_stream : TupleStream out_stream)`

Takes the `in_stream` and performs the project as specified by `cond`.

`cond` is a sequence of attribute names seperated by "AND". Each tuple of `in_stream` is restricted to the specified columns, and then output as part of `out_stream`. EOS is output when EOS is seen on `in_stream`.

Note: this definition of `project` does not specify how it restricts the columns level in the constituent tuples. It potentially could just restrict the valid names that the accesor methods accept and not change the underlying datastructure at all. However this behavior would lead to memory bloat [whenever the tuple dies or is joined all such bloatedness disappears] which might not be desired. It probably is useful to write another `project`, say `project_strict` that uses minimal memory.

- o `select (String cond, TupleStream in_stream : TupleStream out_stream)`

Selects the tuples that match `cond` from `in_stream`.

`cond` is composed of a sequence of select conditions seperated by "AND". A select condition will be of the form "ATTR ? CONST" where "ATTR" is the name of an attribute in the relation, "?" is a comparison operator, and "CONST" is a constant type compatible with the specified attribute. All 6 comparisons are supported.

Output commences with the first tuple from `in_stream` that matches `cond`. EOS is output when EOS is seen on `in_stream`. If no tuples match, then an empty stream will be output [as opposed to no stream]

- o `sort (String attr_name, String sort_type, Relation in_rel : Relation out_relation)`

Sorts `in_rel` based on the attribute `attr_name` in the order specified by `sort_type`.

`sort_type` can be either "ascending" or "descending". `sort` is guaranteed to be asymptotically efficient [$O(n \log n)$]. `out_rel` is output only after the sort has finished.

- o `stream (Relation in_rel : TupleStream out_stream)`

Takes `in_rel` and constructs a stream out of it.

Each row of `in_rel` is output as part of `out_stream`. Row ordering is not necessarily preserved. EOS will be output after all rows of `in_rel` have been output as tuples.

- o `relsprintf (String attr_name, String format, TupleStream in_stream : TupleStream out_stream)`

Applies `format` to each tuple in `in_stream`.

`format` is a String that has "%ATTR%" where the value of attribute `ATTR` should be inserted. The result of this formatting is then written as attribute `attr_name`. "%%" will insert a "%" in the resultant string.

Each tuple is output as it is formatted. EOS is output when EOS is seen on `in_stream`.

- o `union ((TupleStream in_streams)* : TupleStream out_stream)`

Combines the `in_stream`'s into `out_stream`.

`union` takes the tuples found the the vararg input and produces a single tuple stream. All tuples output will share the same attribute list. Duplicate tuples will not be output. Output will commence with the first tuple seen, and EOS will be output when EOS has been seen on all inputs.

- Database Operators

- o `exportrelation (String dest, String name, Relation in_rel :)`

Persists the given `in_rel` to the specified `dest`

`dest` is the connect string for the database that is to hold the relation. **Formatting of this needs to be discussed.** `name` is the name under which to store the relation.

- o `exportrelation2 (String dest, String name, TupleStream in_stream :)`

Persists the given `in_stream` to the specified `dest`

`dest` is the connect string for the database that is to hold the relation. **Formatting of this needs to be discussed.** `name` is the name under which to store the relation.

- o `importrelation (String source, String name : TupleStream out_stream)`

Imports the relation `name` from `source`.

`source` is the connect string for the database that holds the required relation. **Formatting of this needs to be discussed.** `name` denotes the name of the desired relation. Output begins at the earliest moment. `EOS` is output after all of the requested relation have been output.

- o `query (String source, String query_text : TupleStream out_stream)`

Performs the `query_text` on `source`.

`source` is the connect string for the database that is being queried. **Formatting of this needs to be discussed.** `query_text` is a query that will be understood by the database, no parsing of this argument occurs. Output begins at the earliest moment. `EOS` is output after all tuples that satisfy the query have been made part of `out_stream`. An empty stream is output if no tuples satisfy the query.

- o `query2 (String source, String query_text :)`

Performs the `query_text` on `source`.

`query2` works exactly like `query` except that it does not return any result. This is useful for queries like "update", etc.

- Utility Operators

- o `makestream2 (String something_ugly : TupleStream out_stream)`

Transforms `something_ugly` into a stream.

`makestream2` provides for inline construction of a stream. This is most useful for debugging purposes. **Formatting of `something_ugly` needs to be discussed.** In general, `makestream` should be preferred.

- o `constructfunction (String func_name : Function fun)`

Gets the function `fun` from the system.

`constructfunction` will instantiate a requested function from the Registry, and failing that, the local loadpath [classpath in the case of java].

- o `noop (:)`

Does nothing except be an operator.

Note that as an operator this may be useful as a synchronization primitive.

- o `notify (String dest, Stream msg_stream :)`

Takes `msg` and sends each piece to `dest`.

`dest` encodes where to send the message. It is presumed to be a URL [ie 'mailto:theseus@isi.edu']. Each value in `msg_stream` is sent to `dest`.

It might be useful to supply a version that takes only one message.

See also `apply`.

- o `print (Stream in_stream :)`

Prints to stdout `in_stream`.

Each value of `in_stream` is transformed into a string, and then output by the executor on what the plan understands as 'stdout'. Note that this is mostly just useful for debugging as most plans will manage their datasets through mechanisms other than stdout.

- o `sleep (String time :)`

Does nothing for an amount of time equal to `time`

`sleep` is a slow noop. `time` is measured in milliseconds. `sleep` is guaranteed to wait at least as long as `time`. Due to uncertainty in the execution system, it may appear to wait longer.

- o `sprintf (String format, (Object data)* : String out_string)`

Inserts data into the '%s's in `format`.

Each peice of the `vararg` input data is transformed into a string, and then placed in the corresponding part of `format` This produces an `out_string`. Note that a '%%' in `format` will be transformed into '%' in `out_string`.

- Misc Operators

- o `retrieve (String wrapper, String query : Stream out_stream)`

Executes the requested `query` on the specified `wrapper`.

`retrieve` functions just like `importrelation` exception that `wrapper` is interpreted as a connect string to an `ariadne` wrapper.

Formatting of this needs to be discussed.

- o `fetch (String in_url : Stream out_stream)`

Reads data from a web page, identified by a URL, sending the result out as a stream. Typically used in conjunction with `Extract`.

- o `extract (Stream in_stream, String in_rules : TupleStream out_tuples)`

Takes the input data stream (which is XML, HTML, etc) and, based on the extraction rules specified by `in_rules`, extracts information into tuple form, exporting the result as a stream of tuples.

13. Developing Operators

13.1. Writing Queue Processing Functions

As was mentioned in Section 6, the operator processing model assumes the existence of functions that can process the non-streamed and streamed input. Specifically, three types of functions are necessary:

- the per-stream function: a single function (called “on_init”), called once (per transaction) for purposes of processing the per-stream inputs
- tuple functions: called for each tuple of streamed input
- EOS functions: called for each streamed input

For example, if an operator has five inputs – 3 non-streamed and 2 streamed – the operator developer needs to implement 3 non-stream functions (one for each non-stream queue) and 4 streamed functions (a per-tuple function and EOS function for each queue): 7 functions in all.

As far as naming the functions, they must be directly based on the name of the queue, as specified in the operator metadata. Each non-EOS function takes the form of:

```
on_<name of queue>
```

Each EOS function takes the form of:

```
on_<name of queue>_EOS
```

So, for example, let us recall the sample metadata for Select:

```
Operator Select {  
    INPUT:      relation InData, string Condition;  
    OUTPUT:     relation OutData;  
    BLOCKING:   No;  
    PARALLELIZABLE: Yes;  
    GRANULARITY: any;  
}
```

The resulting functions to be coded, in Java, would be:

- on_InData
- on_InData_EOS
- on_init

Each of these functions will pass the relevant data as an argument, although `on_init` will really be called with a generic input argument descriptor. Thus, the operator developer does not need to manually get that data. Also, the data being processed is assumed to belong to a transaction. Although there will be an API to allow the transaction to be determined, it is generally not necessary nor useful for the operator to know or care about which transaction it is processing. So, the function signatures for these functions will be:

```
public void on_InData (StreamingInputArg a_arg)
{
    /* Write tuple processing code */
}

public void on_InData_EOS ( )
{
    /* Handle EOS */
}

public void on_init (InputArgSet a_in, OutputArgSet a_out)
{
    /* Interrogate the InputDesc data structure for its names/value */
}
```

Note that enablements (both consumed and produced) obviously cannot be handled by the operator developer since the enablements are a plan-specific notion. Instead, the system guarantees that the firing rule is not called until the enablements have all arrived and only generates enablements after performing the first firing rule for that transaction.

14. Notes

14.1. Streaming

14.1.1. Shallow Copying of Tuples

To promote flexibility in streaming, Theseus does not queue each tuple, but rather a pointer to a set of tuples. The finest granularity of streaming is thus one where a pointer to one tuple is streamed between operators. Streaming pointers instead of tuples specifically allows two important features:

- **variable streaming rates:** for purposes of optimization, it may be necessary to adjust the streaming rates between operators to a more coarser level.
- **efficient I/O between operators:** instead of (deep) copying the actual tuple data tuples between operators, Theseus performance can benefit from pointer-based (shallow) copying – this is especially useful for operators that do not alter the data, such as Select.

14.1.2. Relation Identifiers

In Theseus, relations are composed of:

- Attribute list: defines the schema for that relation (names and types)
- Tuples: the data itself
- End-of-Stream: end of the relation

When thinking about the processing from an operators point of view, one issue that streaming raises is: when is the attribute list communicated and how does the operator know to associate a streamed tuple or EOS with a given relation and attribute list?

For example, consider a plan with two operators: a Retrieve followed by a Select. Suppose the inputs to Retrieve are both variables at run-time (i.e., both the source and the query are dynamic). Now, suppose that a single transaction presents a relation of 2 rows to retrieve, each representing a different query/source pair to Retrieve. Given a Retrieve that has at least 2 worker threads, this will cause two external queries and extraction to happen at the same time, finishing in some interleaved fashion. What will happen next is that Select will be receiving these tuples in some unknown order. Since they are tuples that belong to the same transaction, what can Select use to (a) distinguish the tuples from each other and (b) ensure the type compatibility of the tuples from each Retrieve (if it gets type-incompatible tuples, it should generate an error)?

The solution to this will be communicate attribute lists once to consumer operators and then to use relation IDs to identify the rest of the tuples for that relation, as well as its EOS. The algorithm that a producer uses when communicating to a consumer:

- Communicate the first “tuple”, which is really a metadata tuple that contains
 - the details of the attribute list
 - unique relation ID
- Communicate the remaining tuples, each tagged with this relation ID
- Communicate the EOS, also tagged with the relation ID

The consumer knows that the first tuple is the “metadata tuple” and does the following:

- Receives the first “metadata tuple”
 - Caches the attribute list and relation ID
- Receives remaining tuples
 - Ensures that their relation ID matches the ones it knows about
 - If not, generates an error
- Receives EOS
 - Calls operator function for EOS processing
 - Flushes the relation ID and metadata from its cache

14.2. Notes on Subplans

14.2.1. Splicing Plans Together

The implementation of non-recursive subplans depends on how Theseus is invoked. If Theseus is invoked from the command line, the system will read in both the parent and child plans before execution, merging them together where appropriate to create a single global plan. If Theseus is being deployed in plan-server mode, two situations are possible. If the plan server is already scheduled to keep both the plan and subplan active, there is no need to merge plans together – the subplan call can simply be replaced by a call to the remote plan. This case is highly optimal when plan servers exist on multiple machines (or when there are multiple processors), since parallelism can be better exploited. However, if the subplan is not scheduled to be active by the plan server, it is merged into the parent plan, just as is the case with the command-line style invocation.

14.2.2. Scaling Worker Threads

With recursive subplans, of course, only one set of operators need be available. However, for purposes of scalability, it may be the case that the pool of worker threads for those operators scales up appropriately, to deal with the expected high levels of concurrent requests. This may be particularly relevant for those operators that are I/O bound, such as Retrieve. Through profiling, the scaling of worker threads could be optimized. But, for now, there will be just be some simple algorithm for scaling. For example, when detecting that a plan is recursive, we could simply triple the number of threads for each operator, theoretically enabling the execution of 3 concurrent frames.

14.3. Notes on State Management

In the context of the Manager/Worker model, one can implement state such that it is managed by the manager thread. This is useful because workers exist within the manager address space (so they can easily access this state) and different workers can safely update state without worrying about inconsistency – the manager thread synchronizes access to this data structure. Furthermore, since every manager controls its own state, the synchronization challenge is not global – instead, it is managed in a decentralized fashion.

To get a better idea of what a state management API might look like, see the example from the `VarUnion` operator implementation, in the Examples section.

15. Examples

15.1. Operator Implementation Example

The following is an example of the code required to develop an operator in Theseus V2. The operator shown is `VarUnion`, which takes a variable number of input `TupleStreams` and outputs a variable number of unioned `TupleStreams` (basically, the outputs are copies of each other).

```
import java.util.*;
import java.net.*;
import java.io.*;

import theseus.api.InputArg;
import theseus.api.InputArgSet;
import theseus.api.Logger;
import theseus.api.NonStreamingInputArg;
import theseus.api.NonStreamingOutputArg;
import theseus.api.OutputArgSet;
import theseus.api.Registry;
import theseus.api.StreamingInputArg;
import theseus.api.StreamingVarInputArg;
import theseus.api.StreamingVarInfo;
import theseus.api.StreamingOutputArg;
import theseus.api.StreamingVarOutputArg;
import theseus.api.QueueType;
import theseus.api.QueueDesc;
import theseus.api.Operator;
import theseus.api.OpProperty;

public class OpTst_varunion extends Operator
{
    private class OpTst_varunion_state
    {
        public StreamingVarOutputArg arg;
        public int tup_count;
        public int EOS_count;
    }
    OpTst_varunion_state state;

    public OpTst_varunion() { }

    public OpProperty[] getProperties()
    {
        return null;
    }

    public QueueDesc[] getInputs()
    {
        return new QueueDesc[]
        {
            new QueueDesc("rel", QueueType.VA_STREAMING)
        };
    }

    public QueueDesc[] getOutputs ()
    {
        return new QueueDesc[]
        {
            new QueueDesc("out_data", QueueType.VA_STREAMING)
        };
    }
}
```

```

public void on_init(InputArgSet a_in, OutputArgSet a_out, Registry a_reg,
    Logger a_log) throws Exception
{
    StreamingVarInfo i = a_in.getStreamingVarInfo("rel");

    state = new OpTst_varunion_state();
    state.arg = a_out.getStreamingVarArg("out_data");
    state.EOS_count = i.getLength();
    state.tup_count = 0;
    createState("state", state);
}

public void on_rel(StreamingVarInputArg a_in)
{
    state = (OpTst_varunion_state)getLockedState("state");
    for (int i=0; i<state.arg.getLength(); i++)
        state.arg.getArgAtPos(i).putObject(a_in.getArg().getObject());
    state.tup_count++;
    unlockState("state");
}

public void on_rel_EOS()
{
    state = (OpTst_varunion_state)getLockedState("state");
    state.EOS_count--;

    if (state.EOS_count == 0)
    {
        for (int i=0; i<state.arg.getLength(); i++)
            ((StreamingOutputArg)state.arg.getArgAtPos(i)).putEOS();
    }
    unlockState("state");
}

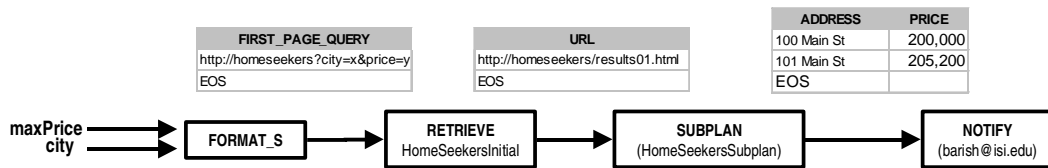
```

15.2. Plan Examples

15.2.1. HomeSeekers

Based on a city and a maximum price, this plan fetches houses from HomeSeekers that meet the specified criteria. Specifically demonstrated in this plan are:

- recursive calls via Subplan
- how the AtLeastOne operator is used



```

PLAN HomeSeekers
{
  INPUT: maxPrice, city
  OUTPUT: houseDetails

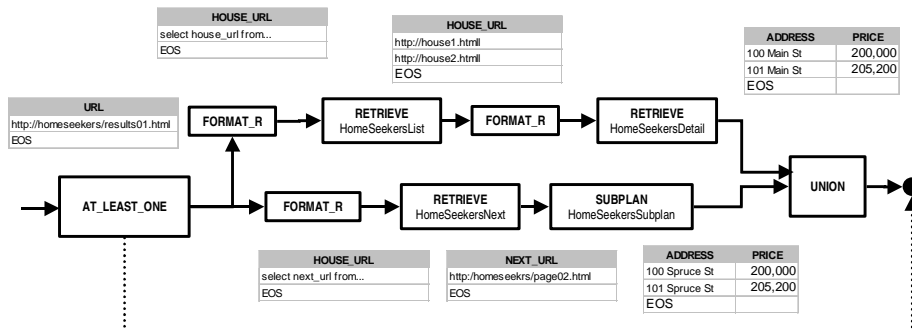
  BODY
  {
    FORMAT_S ("select results_page from HomeSeekersInitial where maxPrice < %s and city='%s'", maxPrice, city : firstPageQuery)

    RETRIEVE (firstPageQuery, "HomeSeekersInitial" : initialPage)

    SUBPLAN ("HomeSeekersSubplan", initialPage : houseDetails)

    NOTIFY ("email", "barish@isi.edu", houseDetails)
  }
}

```



```

PLAN HomeSeekersSubplan
{
  INPUT: candidatePage
  OUTPUT: matchingHouses

  BODY
  {
    AT_LEAST_ONE (candidatePage : validPage, matchingHouses)

    FORMAT_R (validPage, "URL", "select house_url from HomeSeekersList where results_url='%url'" : listQuery)

    FORMAT_R (validPage, "URL", "select next_url from HomeSeekersNext where results_url='%url'" : nextQuery)

    RETRIEVE (listQuery, "HomeSeekersList" : houseUrls)

    FORMAT_R (houseUrls, "HOUSE_URL", "select price, address from HomeSeekersDetail where house_url='%house_url'" : detailQuery)

    RETRIEVE (detailQuery, "HomeSeekersDetail" : rightUnion)

    RETRIEVE (nextQuery, "HomeSeekersNext", : nextPageUrl)

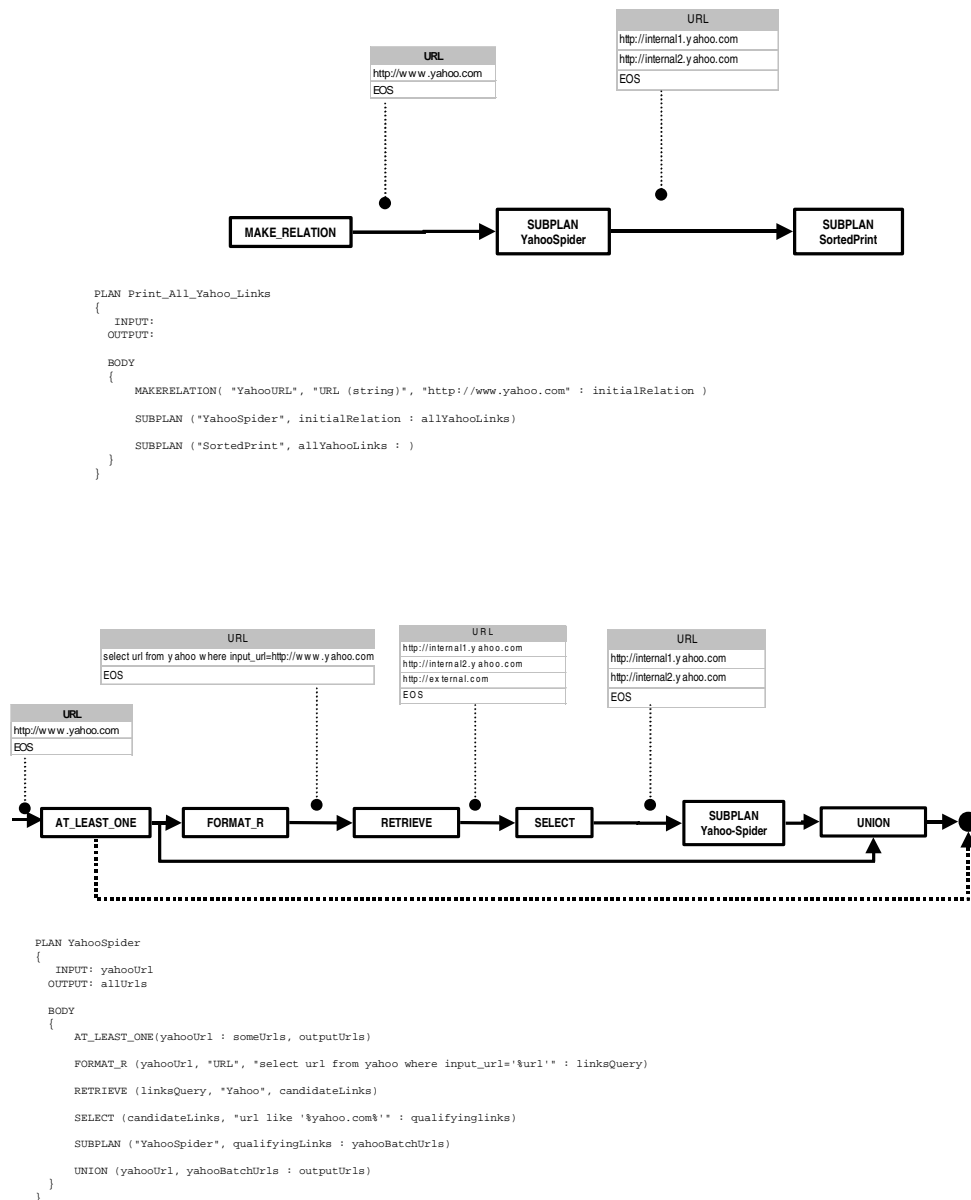
    SUBPLAN ("HomeSeekersSubplan", nextPageUrl : rightUnion)

    UNION (leftUnion, rightUnion : matchingHouses)
  }
}

```

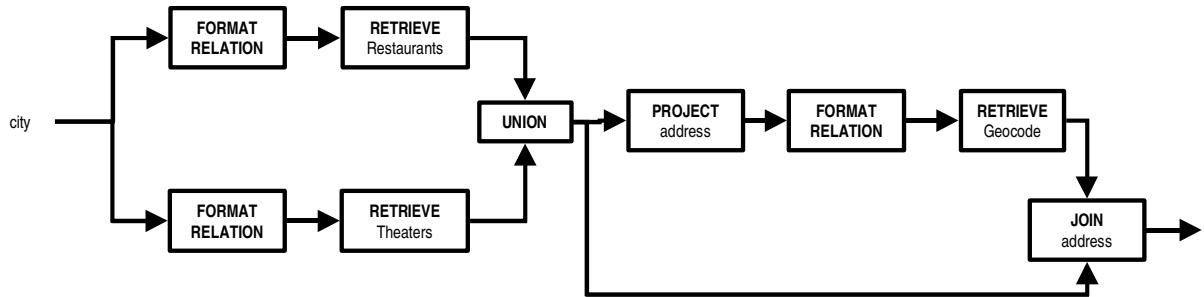
15.2.2. Yahoo Spidering

This plan retrieves, sorts, and prints all of Yahoo's internal links.



15.2.3. TheaterLoc

This plan simply retrieves all of the restaurants and theaters for a given city.



```
PLAN TheaterLoc
{
  INPUT: city
  OUTPUT: restaurants_and_theaters

  BODY
  {
    FORMATRELATION(city, "city", makeRestoQuery : restoQuery)

    FORMATRELATION(city, "city", makeTheaterQuery : theaterQuery)

    RETRIEVE ("cuisinenet", restoQuery : restaurants)

    RETRIEVE ("yahoo", theaterQuery : theaters)

    UNION (restaurants, theaters : places)

    PROJECT(places, "address" : place_addresses)

    FORMATRELATION(place_addresses, "address", makeGeoQuery : geoQuery)

    RETRIEVE ("geocoder", geoQuery : geocoded_places)

    JOIN (places, geocoded_places, "address = address" : restaurants_and_theaters)
  }
}
```