iQdep: Integrated Query, Data and Event Processing

by

Suman Subash

Advised By
Dr. Peizhao Hu
Rochester Institute Of Technology

August 2016
# Contents

List of Figures ................................................. ii
List of Tables ............................................... iii

1 Introduction ............................................... 1

2 Related Work ............................................... 2
  2.1 Dryad .................................................. 2
  2.2 Spark ................................................ 3
  2.3 Naiad ............................................... 4
  2.4 Query Processing in Wireless Sensor networks: ....... 5
     2.4.1 TinyDB: .................................... 5
  2.5 Complex Event Processing: ........................... 5

3 The iQdep Architecture ................................. 7
  3.1 Data Model ............................................ 8
  3.2 Components .......................................... 8
  3.3 Process Model ....................................... 9
  3.4 Distributed Architecture Model ..................... 10
  3.5 Query Processor ..................................... 11
     3.5.1 Query Parser .................................. 11
     3.5.2 Query Optimizer ............................. 14
     3.5.3 Query Planner and Deployment ............. 15
  3.6 Programming Abstraction ............................ 16
     3.6.1 Map .......................................... 17
     3.6.2 Filter ....................................... 17
  3.7 Continuous Queries ................................ 17
     3.7.1 Window ........................................ 18
  3.8 Event Processing ................................... 18
     3.8.1 Events ........................................ 18
  3.9 Integrated Architecture ............................ 20
  3.10 Examples ............................................ 21
     3.10.1 Fire Alarm ................................... 21

4 Future Work And Conclusion ......................... 22
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Dryad Architecture</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Spark Architecture</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Naiad Timely Dataflow</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>High Level View</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>iQdep Components</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Process Model</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Shared Nothing Architecture</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>Query Processor</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>Query Lexer Rules</td>
<td>12</td>
</tr>
<tr>
<td>3.7</td>
<td>Lexer Output</td>
<td>13</td>
</tr>
<tr>
<td>3.8</td>
<td>Parser Grammar Rules</td>
<td>13</td>
</tr>
<tr>
<td>3.9</td>
<td>Parse Tree</td>
<td>13</td>
</tr>
<tr>
<td>3.10</td>
<td>Breakdown Predicates</td>
<td>14</td>
</tr>
<tr>
<td>3.11</td>
<td>Pushdown Predicates</td>
<td>15</td>
</tr>
<tr>
<td>3.12</td>
<td>Pushdown Projection</td>
<td>15</td>
</tr>
<tr>
<td>3.13</td>
<td>Query Deployment Example</td>
<td>16</td>
</tr>
<tr>
<td>3.14</td>
<td>Operators</td>
<td>16</td>
</tr>
<tr>
<td>3.15</td>
<td>Operators Contd</td>
<td>17</td>
</tr>
<tr>
<td>3.16</td>
<td>Integrated Architecture</td>
<td>20</td>
</tr>
</tbody>
</table>
List of Tables

2.1 CEP Literature Review ........................................... 6
Chapter 1

Introduction

Cloud Platforms such as Google Dataflow, Dryad, and Apache Spark is designed primarily for Data centers. They provide acyclic data flow model where data passed through deterministic data flow operators. Furthermore, Cloud Platforms also provide Complex Event processing (CEP) engines which perform complex event pattern analysis to trigger an action.

Although cloud platforms provide powerful Dataflow operators, they aren’t designed for IoT on-device execution. Currently, data from IoT devices are transferred to the cloud platforms where, using the powerful data flow operators, meaningful insights can be obtained. Also, current systems lack tight integration between CEP and Streaming Analytics. Streaming and CEP frameworks are studied individually in literature.

In this paper we propose an on-device execution framework called iQdep for IoT which has the following goals. First, provide a stream processing engine optimized for IoT devices. Second, integrate stream processing engines with complex event processing engines for IoT. Finally, we provide a bridge which provides seamless integration between on-device execution and cloud.

The paper is organized as follows: In section 2, we discuss about related work. Section 3 discusses about the architecture of iQdep. Finally, we’ll end the paper with Future work and conclusion.
Chapter 2

Related Work

2.1 Dryad

Dataflow model was popularized by Dryad [1] in 2006. Using Dryad, users could define a Dataflow graph which is represented as $G(V, E)$, where $V$ is Vertices and $E$ is Edges. Vertices represent sequential programs and Edges represent the communication mechanism. Dataflow graph is modelled using several operators for data processing. Some operators include pointwise, cross product and fork. Using these operators, complex directed acyclic graph (DAG) can be constructed. Although, Dryad model was a generic representation of map reduce, it still had problems solving Iterative Machine Learning Algorithms.

Dryad system overview can be seen in Figure 2.1. Name server (NS) is the resource manager for the cluster. It collects information such as available cores and memory across the cluster. Job manager (JM) uses the collected information to make scheduling decisions for the user submitted job. Each worker node has a Daemon (D) which executes the vertices sent by Job Manager.
2.2 Spark

Apache Spark\cite{2}\cite{3}\cite{4} \cite{5} solved the problems mentioned in Dryad by introducing RDD (Resilient Distributed Datasets). RDDs are immutable partitioned collection of records on which a deterministic set of operators can be applied. Since the operators are deterministic and operate on all the records in the partition, fault tolerance of RDDs can be obtained using Lineage of RDDs. Operators are classified as Transformations and Actions. Transformations are lazy and doesn’t cause execution across the cluster. Actions trigger the Dataflow graph to execute across the cluster. Furthermore, RDDs can be persisted in memory which helps for iterative machine learning algorithms.

Figure 2.2 describes the architecture of Apache Spark in Cluster mode. Apache Spark has a driver program which allows users to interact with the cluster. When the driver program submits the task to the Cluster Manager, it determines which machine the task should be executed on. Worker nodes report the resources to the cluster manager. Additionally, the worker nodes manage the cache which could be used to persist the RDDs.
2.3 Naiad

Naiad\cite{6}\cite{7}\cite{8}\cite{9}\cite{10} works on Directed graph with cycles. Since Naiad allows cycles in the graph, iterative and incremental computations are easy to compute. Although iterative and incremental computations are possible in Apache Spark, it has 1 second latency to perform computations due to its micro-batch architecture. Naiad avoids this by providing cycles to the Dataflow graph.

Naiad provides loops in Dataflow graph with the following constraint- Only within a looping context the cycle can exist. And each looping context should have an ingress, egress, and a feedback vertex. See Figure 2.3
2.4 Query Processing in Wireless Sensor networks:

Broadly, there are two main approaches for data acquisition [11]: pull-based and push-based.

**Pull-based:** In this approach, data is acquired only at fixed user-defined intervals. Some examples of pull-based approaches are TinyDB[12] and Cougar[13].

**Push-based:** In push-based approaches, the sensors values are communicated only when there is a deviation in the expected value. Central Server and Sensors agree on the expected behavior. This approach enables users to identify interesting patterns in sensor values. Some examples of push-based approaches are PRESTO[14] and Ken.

Since iQdep uses Pull based approach, we’ll be discussing about tinydb in more detail.

2.4.1 TinyDB:

TinyDb uses a pull-based approach for query processing. It defines a table called sensors which is partitioned across the sensor devices. Every sensor device can produce one or more sensed value such as Light Intensity and Temperature. TinyDB schema looks as shown below:

\[
\text{Timestamp, } A_1, A_2, A_3, ..., A_n
\]

where \( A_1 \) is first attribute which could be temperature for example. Some devices might not produce all attribute values. In that case TinyDB allows sensors to produce NULL values for certain columns.

For Query processing, TinyDb constructs a Semantic Routing Tree.[12]

2.5 Complex Event Processing:

Complex Event Processing has been widely used in Financial Industry for high performance trading. Complex Event Processing has been well studied in literature but mostly from Temporal Operators perspective. Spatial Operators and integration with Streaming Dataflow still requires further research. For IoT, both Streaming Dataflow Processing and Complex Event Processing(Temporal and Spatial) are necessary. Below Table 2.1 provides a overview of the CEP research:

As we can see from the Table 2.1, none of the papers discuss about Spatial Operators. Temporal Operators have been well discussed in all the papers. Cayuga and Snoop IB
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial Operators</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Temporal Operators</td>
<td>DUR</td>
<td>WITHIN</td>
<td>RANGE</td>
<td>WITHIN</td>
</tr>
<tr>
<td>Nested Queries</td>
<td>YES</td>
<td>Limited</td>
<td>YES</td>
<td>Limited</td>
</tr>
<tr>
<td>Condition Operators</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Integration With Dataflow</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.1: CEP Literature Review

has well defined support for nested queries. Lack of support for integration of streaming Dataflow and complex event processing.
Chapter 3

The iQdep Architecture

In this section, we’ll deep dive into the iQdep Architecture. Figure 3.1 depicts the high-level systems view of iQdep.

A collection of networked devices (IoT) can send data either directly to the cloud systems such as Apache Spark and Google Cloud Dataflow or it could perform on-device execution. The controller is central component which performs tasks such as scheduling and maintaining topologies. The Controller interacts with the Forwarder to build a flow table for Streaming Dataflow. The inputs and output flows are defined in the flow table. Forwarder receives data from other Forwarders, applies a user defined function on the
data, and sends the output to other Forwarders. The bridge enables some part of the
iQdep Dataflow graph to be executed on the cloud platforms.

### 3.1 Data Model

The iQdep framework defines the following data model:

\[ A_1, A_2, A_3, ..., A_n, Time, Location \]

\( A_1 \) to \( A_n \) are sensor attributes such as Temperature and Light Intensity. The sensor
devices expose additional information such as Time and Location. Providing Time and
Location enables us to perform spatial and temporal queries.

### 3.2 Components

The iQdep Framework consists of four main components as illustrated in Figure 3.2.

**Client Query Interfaces:** Client submits user requests to the iQdep framework using
one of the protocols mentioned in the Client Communications Figure 3.2. Direct
Communication is used when the client interacts with the iQdep framework directly. JD-
BC/ODBC is useful while interfacing with external applications such as reporting/visualizations tools. The iQdep framework can also be extended to provide additional client
communication protocols. Currently, iQdep only supports a Direct connection.

**Query Processor:** When the client submits the SQL query, query processor receives
the query and performs the following tasks. First, query parser performs lexical analysis
and parsing to build a query Parse Tree (PT). From the PT, an Abstract Syntax Tree
(AST) is constructed. Next, query optimizer accepts an AST and produces an optimized Query Plan. The optimized Query Plan is mapped to a Physical Plan using the
Topology manager. Finally, the Plan Executor deploys the Physical Plan with the least
cost across the IoT devices.

**Storage:** Several operators in iQdep Query Plan request data from the storage de-
vices. Upon receiving a request, the storage component acquires a lock on requested
data to ensure consistency. Finally, the transactions performed by the buffer manager
will be logged to an external device for fault tolerance. The iQdep does not have a
well-defined storage manager implemented which will be addressed in future work.
Shared Services: During query execution, most of the services mentioned in the Shared Services will be utilized. Memory Manager provides memory allocation and de-allocation APIs during the lifetime of the query.

3.3 Process Model

In order to handle concurrent requests from many users, a good process model needs to be defined. The defined process model will have a significant impact on the overall software architecture. Several process models have been studied in literature and some of them are

- Process per Worker
- Process Pool or Thread Pool
- Thread per Worker
In iQdep framework, we use a variant of the Thread per Worker process model, called CPU Pinned Thread per Worker. This model is also incorporated in MySQL[19] and IBM DB2 [20]. We use Akka Actors[21] to pin a Thread to a CPU core. The Actor is backed by a thread pool executor with a single thread within it. Furthermore, it has a priority mailbox to handle different prioritized messages. The communication between threads happen via asynchronous message passing.

Figure 3.3 describes the CPU core Pinned thread per worker process model. The reason why this process model has been selected is because its easy to implement since each thread is mapped directly to an OS thread. Additionally, its easier to debug when we have single thread per CPU core. Scaling the CPU pinned thread per worker might have challenges especially on IoT devices due to the memory constraints, since each thread consumes around 1MB of memory. We will be addressing this as part of the future work.

### 3.4 Distributed Architecture Model

There are several distributed architectural models studied in literature. Some of them include

- Shared Memory
- Shared Nothing
- Shared Disk
- NUMA: Non uniform Memory access
The iQdep framework uses Shared Nothing architecture because it provides easy fault tolerance management. The device failures are isolated to a single machine in shared nothing architecture thus providing high availability. Finally, with proper replication strategy, we can ensure the system is available at all times.

Figure 3.4 describes the shared nothing architecture, in which the hardware resources such as RAM and Disk, are not shared across devices. Every device is responsible for accessing its local data such as temperature and ambient light. The IoT devices communicate using commodity networking components using asynchronous messages.

The CPU pinned threads accepts tasks and performs the required calculations and returns the results back to the client.

Replication strategy is not available in the current version of the framework.

3.5 Query Processor

A query processor takes a iQdep SQL statement parses it, performs optimizations and converts it into a Dataflow graph. The components of query processor was discussed in section 3.1. Figure 3.5 represents the flow once the query is submitted to when the query is deployed.

3.5.1 Query Parser

Given a SQL query, the query parser performs two tasks lexical analysis and parse tree construction.

Figure 3.6 illustrates few lexer rules defined as part of the iQdep framework.
The output of lexical analyzer, given a SQL query, is as shown Figure 3.7. The output \([@0, 0:5 = \text{'select'}, < 6>, 1:0]\) denotes that this is the first token which starts from character position 0 and ends at position 5. The token text is “select”. The token has 6 characters and is present on line 1.

After the lexical analysis stage the parse tree construction begins. Parser grammar is written using CFG (context free grammar). Figure 3.8 illustrates few CFG rules.

The outcome of the parse tree construction for the below SQL query can be seen in Figure 3.9

\[\text{select} \text{col}1, \text{col}2, \text{col}3 \text{ from } (\text{Event } e1, \text{ Event } e2, \text{ Event } e3)\]
Currently iQdep supports only limited grammar rules and lexer rules. More rules need to be added to support additional operators which will be addressed as part of future work.
3.5.2 Query Optimizer

Query Optimizer is responsible for generating the most efficient Logical Plan. There are three ways to optimize the SQL query studied in the literature: Rule Based Optimizer, Cost Based Optimizer and Adaptive Query Optimizer.

Rule Based Optimizer[22] [23]: applies several rules repeatedly on the Logical Plan (Tree/DAG) to generate new Optimized Logical Plan (Tree/DAG).

Cost Based Optimizer[24]: estimates the cost associated with accessing disk, CPU and network for several Logical Plans and outputs the best Logical plan amongst them.

Adaptive Query Optimizer[25][26]: When execution plan changes mid-query the need for adaptive query optimizer becomes evident. This will become significant especially when we are handling continuous queries.

In the iQdep framework, we chose to incorporate a Rule Based Optimizer. It is easy to implement since each rule will not take more than few lines of code. It provides easy extension since users can extend the available rules set to add their own.

Now, we’ll be discussing the different rules available in the iQdep query optimizer and how the rules affect the Logical plan using examples.

(1) Break up predicates: Composition of predicates is split into simple predicates as shown Figure 3.10.

![Figure 3.10: Breakdown Predicates](image)

(2) Pushdown predicates: is mostly used to minimize the network traffic by filtering records at the source. Predicate push down can be seen in action in the below Figure 3.11.
(3) Pushdown Projection: is also used to minimize the network traffic by filtering records at the source. Projection push down can be seen in action in the below Figure 3.12

3.5.3 Query Planner and Deployment

In Query Planning and Deployment phase, an overlay network is formed. The formation of the overlay network occurs with the help of the Controller. Within the overlay network, a leader is elected. The leader will forward the query to each of its children. Forwarding of the query will happen based on the entry in the Flow table provided by the controller. If a child has multiple parents, it’ll choose the parent with least cost, and returns its value to that parent. Parent nodes will append/partially aggregate the values obtained from children along with its own value, and return it to the leader. Leader performs the final aggregation and returns the result to the client. Note, partial aggregations are applicable only when the query is commutative and associative.
Figure 3.13 demonstrates an example of a Deployed query on IoT devices. An overlay network is formed with a leader elected as the root node. In this case, node ‘A’ receives the query, forwards the query to its children. Each child performs either forwarding or returns the results based on whether its a leaf node.

The parent nodes perform partial aggregation when the query is commutative and associative. Final aggregation is performed at the leader node. Average Operator can perform partial aggregation by reducing the query to return \((\text{sum}, \text{count})\). Leader performs the division of \((\text{Total Sum} / \text{Total Count})\).

### 3.6 Programming Abstraction

The iQdep framework allows the users to write a Dataflow graph using API abstractions. iQdep provides APIs similar to Dryad and Apache Spark.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arguments</th>
<th>Output</th>
<th>Diagram Representation</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>Function(&lt;T, R&gt;)</td>
<td>(\text{AbstractVertex}&lt;R&gt;)</td>
<td>(\stackrel{T}{\longrightarrow} R)</td>
<td>Celsius to Fahrenheit</td>
</tr>
<tr>
<td>filter</td>
<td>Predicate(&lt;? extends T&gt;)</td>
<td>(\text{AbstractVertex}&lt;T&gt;)</td>
<td>(\stackrel{T}{\longrightarrow} \stackrel{\text{F}}{\longrightarrow} T)</td>
<td>(&gt; 30\ \text{F})</td>
</tr>
<tr>
<td>sum(\text{int}), sum(\text{Long})</td>
<td>-</td>
<td>(\text{int, long})</td>
<td>(\stackrel{\text{int}}{\longrightarrow} \stackrel{\text{int}}{\longrightarrow})</td>
<td>Sum of Temperatures</td>
</tr>
<tr>
<td>max(\text{int}), max(\text{Long})</td>
<td>-</td>
<td>(\text{int, long})</td>
<td>(\stackrel{\text{int}}{\longrightarrow} \stackrel{\text{int}}{\longrightarrow})</td>
<td>Maximum Temperature</td>
</tr>
</tbody>
</table>
iQdep APIs can be classified into two categories: Transformations and Actions. Transformations and Actions are listed in Figure 3.14 and 3.15. The dotted circles are Actions.

### 3.6.1 Map

Map is a transformation which accepts a Function as an argument. Function takes an argument of type T and returns a type R. Map allows us to convert, for example, temperature from one unit to another.

### 3.6.2 Filter

Filter is a transformation which accepts a Predicate as an argument. Predicate Lambda expression takes an argument of type T and returns a boolean. The output type of the Filter transform will remain the same since it doesn’t modify the input type. Filter allows us to remove unwanted records. For example, if we want to remove temperatures > 30F, filter can be applied.

There are several other operators available as part of iQdep framework. Please look at the github repository for additional details.

### 3.7 Continuous Queries

Continuous queries (CQs) are long repeatedly running queries against new data to produce new results. The data might be coming from sensors or persistent storage. Since the

---

**Figure 3.15: Operators Contd**
data might be bounded or unbounded, the programming abstractions discussed above will not suffice. Operators such as Joins and Sort will not complete. The operators which require entire dataset are called “Blocking” operators. In order to provide ”Non Blocking” operators, the concept of window is required.

3.7.1 Window

A window represents a snapshot of a finite portion of a stream at any time point. Operators can compute their functions over the window of data. There are two types of windows: Time based and Tuple based. Time based windows as defined by

\[
\text{Window Size } M \text{ time units, Window Slide } N \text{ time units}
\]

Similarly, the Tuple based windows are defined as follows:

\[
\text{Window Size } M \text{ rows, Window Slide } N \text{ rows}
\]

3.8 Event Processing

Event processing is fundamentally different than Stream processing. Traditionally, Event processing and stream processing is studied separately but iQdep framework allows users to combine Event Processing with Stream processing to model applications.

Event processing in general has three components: Event, Condition and Actions.

3.8.1 Events

Events can be either primitive or composite. Primitive events are simple events which the framework can detect natively. Composite Events are composition of primitive events using event operators.

iQdep framework allows users to compose events using the following operators:

- And ”\&”: is a binary operator which allows composition of events \(e_1\) and \(e_2\) as follows \((e_1 \& e_2)\).
- Or ”\lor”: is a binary operator which allows composition of events \(e_1\) or \(e_2\) as follows \((e_1 \lor e_2)\).
- Not ”\neg”: is a unary operator which allows composition of a event as follows \((\neg e_1)\).
• Followed ",," defines the order in which events should occur. Composite event E = \( (e_1, e_2 \land e_3, e_4) \) represents the order where \( e_1 \) happens before \( e_2 \) and \( e_3 \) which happens before \( e_4 \).

Using iQdep programming APIs, composition of events can be achieved using the syntax shown below:

\[
\text{select } * \text{ from (Event1 } e_1, \text{ Event2 } e_2 \land \text{ Event3 } e_3) \text{ where } e_1.\text{temp} = e_2.\text{temp}
\]
3.9 Integrated Architecture

Stream Dataflow processing with continuous operators are useful when we need to perform data transformations. But when the usecase demands Temporal correlation of events it is difficult to model using Stream Dataflow processing. Temporal Correlation of events is really important for IoT applications. Hence we have Complex Event Processors. Figure 3.4 shows an integrated architecture for Streaming Dataflow and Triggers.

In Figure 3.16, there are three stages: Streaming Dataflow, Event Generation, and Triggers and Rules Manager.

First Stage involves data transformations/aggregations of streaming data coming from either external or internal sources. We use programming abstractions defined in Section 3.2, 3.3 to perform data transformations/aggregations. The result of the streaming dataflow are sent to the Event Generation Stage (Stage 2).

In Stage 2, the output results are converted to Events. A single Event Generation source could output multiple events. Each event generated is sent to the event buffer in Stage 3.

In Stage 3, each event stored in the event buffer is sent to multiple event graphs. The Event graphs are modelled using the Event operators defines in Section 3.4. Each event graph has interval conditions and also has associated rules. If each rule and event graph conditions are met then certain actions are taken.

![Figure 3.16: Integrated Architecture](image_url)
3.10 Examples

3.10.1 Fire Alarm

Given:

- Temperature Sensor collects information for every 5 seconds.
- Smoke Sensor collects information for every 5 seconds.

Requirement: If temperature increases by 20% in 30 seconds window and smoke increases by 10% in 30 seconds window and both occur at the same location and within 1 minute duration then trigger fire alarm and notify user and authority.

Modelling:

$E_1 = \text{temperature increases by 20\% in 30 seconds window}$

$E_2 = \text{smoke increases by 10\% in 30 seconds window}$

$E = E_1 \land E_2$ where one event occurred at time $t_1$ and other occurs at time $t_2$. And $t_2 - t_1 \leq 1 \text{ minute.}$
Chapter 4

Future Work And Conclusion

Optimizations for complex event processing needs further research. Currently, iQdep discusses about spatial operators in terms of k-nearest neighbors and radius for IoT devices; but higher level abstractions for spatial operators are required. Spatial Metadata management will also be part of future work. Further investigation is required on cyclic Dataflow operators which enables low latency querying.
Bibliography


