Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Pattern Matching over Sequences of Rows in a Relational Database System

Master Thesis

Nihal Dindar
Systems Group, ETH Zurich
Department of Computer Science
dindarn@student.ethz.ch

September 30, 2008

Supervised by: Prof. Nesime Tatbul

Abstract

In Complex Event Processing (CEP) applications such as supply chain management and financial data analysis, the capability to match patterns over data sequences is increasingly becoming an important need. This not only involves finding event patterns of interest on live data streams but also requires a similar functionality over archived sequences of streams for historical analysis, verification, and correlation.

The goal of this thesis is to extend a relational database system with the capability to match patterns over contiguous sequences of rows stored in a database table. More specifically, we have implemented a major subset of the 2007 ANSI standard proposal on adding MATCH_RECOGNIZE clause to standard SQL on top of the MySQL open-source database engine. We have done this in a way to leverage the existing MySQL architecture and processing model as much as possible, while at the same time carefully identifying the parts where brand new extensions were necessary. Thus, one of the main contributions of this thesis is that it clearly shows what it takes in general to add pattern matching capability to any relational database engine.

Contents

1 Introduction **5**
- 1.1 Motivation and Scope 5
- 1.2 Thesis Contribution 5
- 1.3 Thesis Organization 6

2 Related Work **7**
- 2.1 Pattern Matching in Data Streaming Systems 7
- 2.2 Pattern Matching in Relational Database Systems 8

3 Language **9**
- 3.1 2007 ANSI Standard Proposal on Adding MATCH-RECOGNIZE to SQL 10
  - 3.1.1 Input Specification 11
  - 3.1.2 Output Specification 11
  - 3.1.3 Pattern Definition 12
  - 3.1.4 Future Pattern Start Specification 12
- 3.2 Implemented MATCH-RECOGNIZE Features 13
  - 3.2.1 Input Specification 13
  - 3.2.2 Output Specification 13
  - 3.2.3 Pattern Definition 13
  - 3.2.4 Future Pattern Start Specification 13
- 3.3 MATCH-RECOGNIZE vs. Other Pattern Languages 14
  - 3.3.1 SASE+ 14
  - 3.3.2 Cayuga 16

4 Implementation **18**
- 4.1 Programming Environment 18
- 4.2 Core Data Structures 18
  - 4.2.1 Finite State Machine 18
    - 4.2.1.1 Construction of FSM 18

4.2.1.2 FSM Execution Status 20
4.2.1.3 FSM Runtime State 23
4.2.2 Match Buffer 24
4.2.3 Input Holder and Input Buffer 24
4.3 Use Case 26
4.4 Statistics 32

5 Architecture **34**
5.1 MySQL Architecture 34
5.1.1 Interaction of the Core Modules 36
5.2 Pattern Matching Module Extensions 37
5.2.1 Parser 37
5.2.2 Select 40
5.2.3 Optimizer 40
5.2.3.1 JOIN::Prepare() 40
5.2.3.2 JOIN::Optimize() 41
5.2.3.3 JOIN::Exec() 41
5.2.3.4 JOIN::Cleanup() 43
5.3 Summary 43

6 Conclusions & Future Work **45**

A EBNF of MATCH_RECOGNIZE **48**

List of Figures

3.1 Comparison of MATCH_RECOGNIZE Functionalities with SASE+ and CEL . . 16

4.1 Class Diagram of MATCH-RECOGNIZE Implementation 26
4.2 Time vs. Price Graph of Stock Table Data 27
4.3 FSM 28
4.4 FSM Execution 30
4.5 Query result on data graph 32

5.1 High-level view of MySQL modules[9] 35
5.2 Parse Tree of Pattern Variable D 39

List of Tables

4.1 FSM Properties 19
4.2 Edge types 20
4.3 Ticker table 21
4.4 Execution of the example query for matching mode ALL 22
4.5 Execution of the example query for matching mode INCREMENTAL 23
4.6 Execution of the example query for matching mode LOCAL MAXIMAL 24
4.7 Execution of the example query for matching mode MAXIMAL 25
4.8 Stock Table 27
4.9 Predicates of FSM edges 28
4.10 Query result for sliding strategy next row 31
4.11 Query result for sliding strategy past last row 31
4.12 Source Lines of Code 33

5.1 Summary of Pattern Matching Extensions on MySQL 44

Chapter 1

Introduction

1.1 Motivation and Scope

Finding patterns over sequences of data is essential to many applications, including analysis of stock market prices [11], RFID-based inventory management [14], health care systems [3], and sensor-based monitoring [4]. Most of these applications require some form of complex pattern analysis on the data. An example to a complex pattern is looking for price increase in a stock of a company followed by a price decrease which is at least 30 percent of the increase.

Pattern matching over sequences of rows is motivated by the high interest in finding patterns in Complex Event Processing (CEP) applications. Finding event patterns of interest is required on live data streams as well as over archived sequences of streams for historical analysis, verification, and correlation. Current SQL standard does not provide correlation among the rows natively. Therefore, extension of SQL is needed for pattern matching.

This master thesis aims to find patterns on contiguous sequences of rows in a database table. It is assumed that the table stores large sequences of time-ordered historical streams. We have added a novel functionality to an existing open-source relational database engine (MySQL). We have done this in a way to leverage the existing MySQL architecture and processing model as much as possible and at the same time, keeping our extensions as modular as possible. The pattern matching functionality currently works end-to-end, from accepting the query in SQL all the way until producing the match results. We have not yet implemented any optimizations, but we have identified potential sources of performance bottlenecks that we are planning to tackle as part of our future research.

1.2 Thesis Contribution

While design and optimization of query languages for pattern queries in database sequences were addressed by previous work [13], Finite State Automata(FSM) query model for relations was not suggested. Extensions to the relational database engine architecture are needed in order

to provide pattern matching functionality. On the other hand, efficient evaluation of pattern matching queries over streams has been a recent topic in the streaming community [5, 3], but the focus has mainly been on streams, which have different behaviour than stored relational tuples.

The main contribution of this thesis is that it clearly shows what it takes in general to add pattern matching capability to a relational database engine.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 reviews related work, including pattern matching in streaming and relational engines.
- Chapter 3 describes the language that we used for pattern matching.
- Chapter 4 provides implementation details.
- Chapter 5 presents our architecture extensions to MySQL.
- Finally, in Chapter 6, we conclude and indicate some possible future work.

Chapter 2

Related Work

There is some previous work about pattern matching. While some of these concentrate on language extensions needed for pattern queries, others also discuss about efficient evaluation model of pattern queries. The previous work falls into two broad classes:

2.1 Pattern Matching in Data Streaming Systems

Pattern matching ability over streams has become an interesting functionality both in research projects and commercial products. We will discuss two research projects, SASE+ and Cayuga, and a commercial product, Coral8.

SASE+ is a complex event language that supports Kleene closure over streams [7]. Different than the approach we used, SASE+ is an agile language and concentrates only on pattern matching on streaming data. In spite of these main differences, the evaluation model of pattern matching queries proposed by this work can be used in general [3]. A special NFA is used as evaluation model. We followed part of the evaluation model of SASE+ in this thesis. Different than SASE+, we performed the integration of the evaluation model to the existing relational engine MySQL.

Another system which supports pattern matching is Cayuga. Cayuga is a stateful pub/sub system based on nondeterministic finite state automata (NFA) for scalable event processing [6]. Cayuga uses SQL like language CEL (Cayuga Event Language) based on Cayuga Algebra [5] to express the queries [6]. Unlike SASE+, Cayuga merges all the FSM states in order to optimize the execution of multiple queries.

Coral8 is a commercial engine which provides Complex Event Processing and it is programmed via the Continuous Computation Language, CCL. CCL is based on SQL, but it also provides pattern matching with MATCHING clause [1]. With MATCHING clause, only the simple relation "followed by" can be modeled. There is no support for more complex event patterns that require regular expressions.

2.2 Pattern Matching in Relational Database Systems

The 2007 ANSI standard proposal on adding MATCH-RECOGNIZE clause to standard SQL is the main related work we used in this thesis. This work proposes new SQL functionality for finding patterns in sequences of rows [8, 2]. We used parts of this language proposal in our work. The language issue is discussed in detail in Chapter 3. Although the proposal clearly defines the syntax and semantics for MATCH-RECOGNIZE, the evaluation model of pattern queries and how to extend that on a relational database are not provided.

Another main proposal about language extensions on SQL for pattern queries is SQL-TS (Simple Query Language for Time Series) [12]. SQL-TS is a construct for specifying complex sequential patterns. Unlike the standard language proposal, the evaluation model and optimization of the SQL-TS queries is also discussed [13]. Extension to the well-known text search algorithm by Knuth, Morris, and Pratt (KMP) is introduced in order to optimize SQL-TS queries. In the way it considers pattern queries as semantic window and modelling them as Finite State Machines, this thesis differs from the SQL-TS.

Chapter 3

Language

In order to express and process patterns on sequences of data, support for regular expressions is needed. Existing SQL standard provides limited support for regular expressions with the LIKE clause. LIKE is used to look for string patterns in a row. The following query is an example for the LIKE clause. Given a table Customers(String FirstName, String LastName, String Email, String Phone):

```
SELECT *
FROM Customers
WHERE Phone LIKE '416%'
```

The above query is looking for customer records whose phone numbers start with '416'. In another words, it seeks for the pattern within the context of a single column, Phone. Therefore, scope of the pattern is one row. For our purpose, it is needed to look for correlations among rows. This cannot be expressed with the LIKE clause and extensions to the SQL syntax are needed.

Language plays an important role in mapping user needs to system functionalities. It is clear that a good language should be both powerful enough to express the needs of the user properly as well as syntactically and semantically clear enough to avoid any ambiguity. By keeping this criteria in mind, we looked at major existing pattern matching languages [2, 6, 7] and we decided to follow the 2007 ANSI standard proposal on adding MATCH-RECOGNIZE clause to standard SQL [8, 2]. We implemented an essential subset of it in MySQL. Though this proposal has some missing functionalities, it is the most complete proposal. A detailed comparison of these languages can be found further in Section 3.3 of this chapter.

In the remainder of this chapter, we first present the proposed language standard and the major subset of this language that we currently support, and then provide a comparison of this language with several other alternatives.

3.1 2007 ANSI Standard Proposal on Adding MATCH-RECOGNIZE to SQL

2007 ANSI standard proposal on adding MATCH-RECOGNIZE clause to SQL [8, 2] proposes a new SQL functionality for finding patterns defined as regular expressions over sequences of rows. New pattern recognition clause, MATCH-RECOGNIZE is introduced which can be applied to tables. It uses the SELECT-FROM-WHERE structure of SQL and adds MATCH-RECOGNIZE to the FROM clause of the query. The syntax overview of the proposal is as follows:

```
SELECT <select-list>
FROM <table-name> MATCH_RECOGNIZE(
   PARTITION BY <field-name>
   ORDER BY <field-name>
   MEASURES <measure-list>
            MATCH_NUMBER
            CLASSIFIER
   ONE/ALL ROW PER MATCH
   MAXIMAL/INCREMENTAL MATCH
   AFTER MATCH SKIP TO NEXT ROW/PAST LAST ROW/...
   PATTERN (....)
      DEFINE <define-alphabets>
)
```

MATCH_RECOGNIZE clause has four main sections: input specification, pattern definition, future pattern start specification, and output specification. Before explaining the details of the each section, let us illustrate the usage of language with an example.

Example: Let Stock(Symbol, Tstamp, Price) be an append-only table with three columns representing historical stock prices. We would like to detect the following pattern in price: a falling price, followed by a rise in price that goes higher than the price when the fall began.

```
SELECT a_symbol, a_price, d_price, match_no
FROM Stock MATCH RECOGNIZE (
   PARTITION BY Symbol
   MEASURES A.Symbol AS a_symbol,
            A.Price AS a_price,
            D.Price AS d_price,
            MATCH_NUMBER AS match_no
   ALL ROWS PER MATCH
   AFTER MATCH SKIP TO NEXT ROW
   MAXIMAL MATCH
   PATTERN (A B+ C* D)
      DEFINE /* A defaults to True, matches any row */
             B AS (B.Price <= PREV(B.Price))
             C AS (C.Price > PREV(C.Price) AND C.Price <= A.Price)
             D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
)
```

3.1.1 Input Specification

Input specification is the part where user can specify the operations on input data on which patterns will be searched. It provides two optional clauses: PARTITION BY and ORDER BY. With ORDER BY clause, user can specify whether the table needs to be ordered by a given set of columns before the pattern search. Actually, the query runs on sequences of rows, which means there is an inherent order in input data. This clause allows to explicitly define that order according to certain fields. The second option, PARTITION BY is used to partition the table according to a given set of columns name.

3.1.2 Output Specification

Output specification is the part where the format and number of the exported columns of the matched pattern are defined. Regarding to the number of result rows, either all the rows of each pattern (with ALL ROW PER MATCH option) or a summary of each pattern (by using ONE ROW PER MATCH option) can be displayed. In the case of ONE ROW PER MATCH option, the summary of the match has to be defined as part of the MEASURE clause by using aggregation functions on variables. MEASURE clause is the place where the format of the output is defined by using the projection and aggregation over variables similar to SELECT clause. Unlike SELECT clause, there are two special result columns which can be defined as part of the MEASURE clause: MATCH_NUMBER and CLASSIFIER. With the MATCH_NUMBER feature, user can declare an exact numerical result column to get the sequential number of the

match. Furthermore, CLASSIFIER allows user to declare a string result column which displays the corresponding variable name in the PATTERN that the row is matched. CLASSIFIER can only be used with the ALL ROW PER MATCH feature.

3.1.3 Pattern Definition

This part has two main functionalities: definition of the pattern as regular expression and specification of the match mode.

PATTERN component is used to specify the regular expression. Variable names are used in the regular expression, each one corresponding to zero, one, or many rows based on given operator. Some of the operators supported in the PATTERN component are: * (0 or more rows), + (1 or more rows), ? (0 or 1 row). For the example query above, A means exactly one row, B means 1 or many rows, C means 0 or more rows, and D means again exactly one row. Therefore, for this example a match contains at least three rows. DEFINE clause is used to attach predicates on these rows in order to specify the correlation between the rows. In addition to the traditional predicate expressions, DEFINE part introduces new operators to correlate the rows: PREV, FIRST, LAST, aggregation, and field access of a match. PREV is used to compare a column of a match with the column of the match before itself. FIRST & LAST are used to access the first and the last row of the groups of rows matching group variables like A*. In addition to these, aggregation operators on group variables can be used like AVG(A.Price) where A is a group variable. Finally, it is also possible to access columns of the matched rows like D.Price > A.Price.

Match mode is also specified in the pattern definition part of the MATCH-RECOGNIZE clause. User can specify whether she/he is interested in the longest sequence of matches by specifying MAXIMAL MATCH, or matches evaluated incrementally row by row by specifying the INCREMENTAL MATCH [2]. In addition to the MAXIMAL and INCREMENTAL match modes, there is also one internal match mode: ALL MATCH. ALL MATCH is the base of the both INCREMENTAL and MAXIMAL matches and reports all possible matches, but according to the original proposal it is not part of the syntax. Therefore, user can not specify the ALL MATCH mode in the MATCH-RECOGNIZE clause.

3.1.4 Future Pattern Start Specification

Future pattern start specification specifies the starting point of the next pattern search after a match is found. For example, in case of non-overlapping patterns are wanted, AFTER MATCH SKIP PAST LAST ROW should be specified, where pattern search starts with the row after the last row in the match. Another option is AFTER MATCH SKIP TO NEXT ROW, which starts to search for new pattern with the row after the first row of the previous match. Other than these two main methods, the starting point of the next match search can also be defined

by giving variable names like AFTER MATCH SKIP TO <variable name> [2].

3.2 Implemented MATCH-RECOGNIZE Features

In this thesis, we picked an essential subset of the MATCH-RECOGNIZE clause and implemented these features in MySQL. Next, these implemented functionalities in each part of the language are discussed.

3.2.1 Input Specification

We implemented neither ORDER BY nor PARTITION BY. Regarding to order, we made an assumption that our data is already ordered by time, since they are stream archive.

3.2.2 Output Specification

We implemented CLASSIFIER and MATCH_NUMBER clauses of MEASURE which are important to understand the relation between the result and the defined pattern. Match numbering is performed based on the order of the first row in a pattern. Furthermore, we implemented ALL ROWS PER MATCH.

3.2.3 Pattern Definition

We implemented singleton variables and group variables (*, +) in the PATTERN clause. In the DEFINE clause, traditional predicates, correlation predicates, and the PREV operator are implemented. These are required to express meaningful queries. As matching mode, in addition to the all of the match modes provided by the standard proposal [2], we introduce a new matching mode: LOCALMAXIMAL. In our opinion, this new match mode is needed to be able to support semantic windows. In LOCALMAXIMAL match mode, the longest sequence of the match in a window is reported. This is different from the MAXIMAL MATCH. MAXIMAL MATCH is looking for the longest match among the windows, which can be thought of as a global maximal match. For sample executions of different match modes, see 4.2.1.2.

3.2.4 Future Pattern Start Specification

The two major options: AFTER MATCH SKIP TO NEXT ROW and AFTER MATCH SKIP PAST LAST ROW are implemented. While TO NEXT ROW clause offers the sliding window behavior, SKIP PAST LAST ROW offers the tumbling window behavior.

3.3 MATCH-RECOGNIZE vs. Other Pattern Languages

There are some other related works that propose alternative pattern matching languages where the pattern is again defined as a regular expression. In this section, we compare MATCH-RECOGNIZE against two such languages: Cayuga [6] and SASE+ [7]. Unlike the MATCH-RECOGNIZE proposal, both of these languages are developed to define patterns on streaming data sequences.

3.3.1 SASE+

SASE+ is a complex event language that supports Kleene closure over streams [7]. Although SASE+ is an agile language and concentrates only on pattern matching on streaming data, the pattern matching properties of SASE+ can be used in more general contexts. One of the important functionalities that SASE+ offers is pattern selection specification. SASE+ allows different pattern selection strategies: Strict Contiguous, Partition Contiguous, Relevant Event Selection (skip till next match), and Exhaustive Pattern Matching (skip till any match). As it can be understood from its name, in contiguous matching option, variables need to strictly follow each other. In case of partition matching, this condition is relaxed, so that variables need to strictly follow each other inside a partition. On the other hand, in "skip till next match" strategy, the irrelevant events are ignored. In other words, the condition is more relaxed in a way that "followed by" is enough regardless of what happens between two relevant events. Last, in "skip till any match" strategy, the condition is even more relaxed and all possible matches are reported. Different then 'skip till next match', even relevant events might be ignored in this option. Let us illustrate the different pattern selection strategies with an example.

Example: Assume that the pattern is defined as A+B+ where mutually exclusive predicates are attached to both A and B. The table T, where the pattern is searched in, has the following sequence of rows:

T = [a1, a2, a3, b1, c3, b2, b3, b4, d3, a3, a5], where a1 means the first row that satisfy the condition defined for variable A. The sliding strategy is SKIP PAST LAST ROW, so matches are not going to overlap and each record is processed only once. First found match at different selection strategies is as follows:

1. Strict Contiguous Match: [a1, a2, a3, b1]
2. Partition Contiguity: It is skipped since it is trivial (same as strict contiguity except it is applied in a partition of the table based on a partition criteria).
3. Skip Till Next Match: [a1, a2, a3, b1, b2, b3, b4] (c3 is ignored since it is an irrelevant record).
4. Skip Till Any Match: [a1, b1] or [a1, a2, b1, b2] or [a1, a3, b2] ... Any of the subset of the table records which satisfy the condition of 'a' followed by 'b' can be the first match. In

this match mode, valid tuples can be ignored like in match 'a1, a3, b2' where the relevant tuples a2 and b1 are ignored.

The first strategy is the main strategy in MATCH-RECOGNIZE proposal. Regarding to the second strategy, PARTITION BY clause in MATCH-RECOGNIZE allows to support. However, there is no direct support for the third and fourth strategies. User can have similar functionality with the third strategy by defining the irrelevant rows at PATTERN clause, whose predicate is the negation of the wanted row. This can be achieved by defining some query rewrite rules. As an example, let us rewrite the pattern definition above with MATCH-RECOGNIZE clause in order to have similar behavior as skip till next match.

Example: A+ can be rewritten as A(A* X*)* where X represents an irrelevant tuple. With a similar idea B+ can be rewritten as (B* Y*)*B where Y represents an irrelevant tuple. The order of relevant and irrelevant tuples is different for two variable, because A is a start variable and pattern has to start with a relevant tuple, and B is the last variable and pattern has to end with a relevant tuple.

```
PATTERN(A1 (A2* X*)* (B1* Y*)* B2)
DEFINE A1 AS a_condition
       A2 AS a_condition
       X  AS !a_condition and !b_condition
       B1 AS b_condition
       Y  AS !b_condition
       B2 AS b_condition
```

With this method, the same functionality can be supported, but in this case X and Y rows which are irrelevant have to be also part of the pattern and be stored as partial matches during the execution which is not desired. In addition to that, user should also worry to eliminate the X and Y rows from the result of the query at MEASURE clause by applying proper projections. Finally, the fourth method can be supported by the proposal which requires ignorance of the relevant tuples with similar query rewrite approach. Different from 'skip till next match', the predicate attached to the irrelevant tuple should be 'true', which enables the ignorance of even relevant tuples. The equivalence of the SASE+ pattern A+B+ with 'skip till any match' strategy can be expressed as follows:

```
PATTERN(A1 (A2* X*)* (B1* Y*)* B2)
DEFINE A1 AS a_condition
       A2 AS a_condition
       X  AS true
       B1 AS b_condition
       Y  AS true
       B2 AS b_condition
```

Although 'skip till any match' behavior can be provided with query rewrite, the same concerns as 'skip till next match' strategy is also valid for 'skip till any match' strategy. On the other hand, relevant event selection strategy can also be considered an important feature of the pattern search, so that it requires to have explicit keyword and be supported natively instead of pushing the responsibility to the user. Finally, non-contiguous record selection strategy can be considered as a missing functionality of the MATCH-RECOGNIZE clause.

3.3.2 Cayuga

Cayuga is a stateful pub/sub system based on nondeterministic finite state automata (NFA) like MATCH-RECOGNIZE. It uses SQL like language CEL (Cayuga Event Language) based on the Cayuga Algebra [5] to express the queries [6]. Similar to MATCH-RECOGNIZE, CEL also allows pattern definition at FROM clause of the query and introduces two additional operators to existing relational operators: NEXT and FOLD. NEXT is a conditional sequence and FOLD is a Kleene Closure. The table in 3.1 compares MATCH-RECOGNIZE with SASE+ and CEL. Since the input specification is considered as heavy process in streaming data, both SASE+

			CEL	SASE+	MATCH_RECOGNIZE
Input Specification*		Partition by	0	0	1
		Order by	0	0	1
Pattern Definition	Regular Expression	Sequence	1	1	1
		Negation	1	1	1
		Kleene+	1	1	1
		Kleene*	0	0	1
	Predicate Specification	Running Aggr.	1	1	1
		Correlation	1	1	1
	Event Selection Strategy	Strict contiguity	0	1	1
		Partition contiguity	0	1	1
		Skip till next match	0	1	not natively
		Skip till any match	0	1	not natively
	Match Mode		0	0	1
Output Specification	Format Specification	Projection	1	1	1
		Final Aggr.	1	1	1
		Composability	1	1	not specified
Future Pattern Strategy	Window Types	Sliding	1	1	1
		Tumbling	0	0	1

Figure 3.1: Comparison of MATCH_RECOGNIZE Functionalities with SASE+ and CEL

and CEL has limited support for it. SASE+ allows to search pattern in a partition of the data as discussed before. Another operation which can be performed on stream data is ordering. Regarding to the order, streams already have an inherit time order and both SASE+ and CEL do not allow to order data based on set of columns different than time. Related to the pattern

definition, the main missing functionality is the selection strategy as discussed early in this section. Regarding to the output specification, the composability is not discussed explicitly in MATCH_RECOGNIZE proposal, but since it is an extension to SQL, composability can be inherited from SQL by nesting the queries. On the other hand, the idea of future pattern selection strategy is not discussed in related work [5, 6, 15, 7], because both of them slide by 1.

In conclusion, the MATCH_RECOGNIZE construct encapsulates most of the functionalities offered by other pattern matching engines. Additionally, it is an extension to SQL, which provides the reuse of the SQL constructs and familiarity to increase usability.

Chapter 4

Implementation

One of the important part of this thesis is the implementation of MATCH-RECOGNIZE clause as part of a relational database system (MySQL). In this chapter, we describe the implementation details.

4.1 Programming Environment

The version of MySQL source code extended for pattern matching is mysql-6.0.3-alpha. As MySQL source code was written in C/C++, C++ is used for the implementation and it is built on a Linux operating system (Ubuntu 8.04). gcc version 4.2.3 is used as the compiler.

4.2 Core Data Structures

Internal representation of MATCH-RECOGNIZE clause is a Finite State Machine (FSM) with Input holder and Input and Match buffers. The following sections will explain these core data structures: FSM, match buffer, input buffer.

4.2.1 Finite State Machine

The main data structure which represents MATCH-RECOGNIZE is a Non-deterministic Finite State Machine. Patterns being defined as regular expressions makes FSM a natural choice.

4.2.1.1 Construction of FSM

An FSM is created during parsing of a given query, when MATCH-RECOGNIZE keyword is seen. Unlike a conventional FSM, our FSM additionally contains required information related to the query such as match mode, sliding strategy, predicates, etc. During parsing, this information is captured from the query and added to the FSM specification. Table 4.1 shows the internal representation of the language constructs in parsing order.

Language Construct	**Internal Representation**	**Comments**
CLASSIFIER	bool classifier true	the value of classifier needs to be set during execution.
MATCH_NUMBER	int match_number = 0	the value of match_number needs to be updated & set in execution.
AFTER MATCH ...	sliding field is set	Input buffer is mainly responsible for this functionality (see section 4.2.3).
... MATCH	strategy field is set	field report_result is also used to implement this functionality (see section 4.2.1.2).

Table 4.1: FSM Properties

Execution model of non-deterministic Finite State Machine specification starts with parsing of PATTERN clause, where states of FSM are created based on variable types. Kleene+ and Kleene* variables are called group variables, while non-group variables are called single variables. The transformation of variables to states are performed in the following way:

- For each single variable, single state is created [3].
- For each Kleene star variable, one star state is created.
- For each Kleene plus variable, one single state and one plus state is created [3].

After creation of variable states, a rightmost final state is added to the FSM. State structure consists of the name of state, the flag indicating whether state is final or not, and set of its edges.
Creation of the edges is performed while parsing DEFINE clause in the following way:

- For each single variable, begin edge with the predicate specified in the query is created [3].
- For each group variable, take edge with the predicate specified in query is created, and forwarding edge with predicate values set to true is created [3].

For each edge in FSM, there are two parameters: input consumption and state change. If input consumption is true for an edge, FSM executes the next tuple in the following turn. Begin edge is an example of edges which consume the input. Assume single state s_1 is followed by another state s_2 and s_1 runs on a record r_1. In this case, based on whether r_1 satisfies the predicate on begin edge, next record r_2 will run on either edge of s_1 or edge/s of s_2. So, r_1 is processed just

once for each FSM instance in this case. If the predicate of input consumption edge is satisfied, then the record will be added to the match buffer as partial match. On the other hand, proceed edge is an example of edges which do not consume the input. The second parameter of an edge is state change, while loop edges do not change the current state of FSM, the proceed and begin edges change the current state of FSM. Table 4.2 shows these edge types in detail.

Edge Name	**Consume Input**	**Move State**	**States**
begin	1	1	forwarding edge of single states
take	1	0	looping edge of group states $(+, *)$
proceed	0	1	forwarding edge of group states

Table 4.2: Edge types

Edge structure stores id, pointers to its source and destination state and its void* predicate. Actual type of predicates is Item tree, which is a MySQL construct to store the expressions. In this way, MySQL structure is reused and the same expressibility as MySQL expressions are provided.

Before concluding the FSM construction, we would like to mention about the flag match_found of FSM which is set when FSM reaches the final state.

4.2.1.2 FSM Execution Status

Each FSM is modeled like a semantic window. Semantic window is a window, which is not defined based on a number of tuples or a time interval but rather semantically, based on the occurrence of some (simple or complex) events [10]. Each tuple evaluated in a window can

- continue the matching process, called PartialMatch
- extend the match, called OpenFullMatch
- terminate the window with a non-match, called NonMatch
- terminate the window with a match, called ClosedFullMatch

Therefore, the FSM can be in one of these four different execution states. The evaluation of the window has three aspects: window size, match reporting, and match buffer action. Among these, the match reporting is based on matching mode. In the following, the effect of each execution state on window is stated.

- In case of PartialMatch, window is extended with the next record.
- In case of OpenFullMatch, window is extended with the next record.

- In case of NonMatch, partial matches are cleaned and window is slided by one.
- In case of ClosedFullMatch, window is slided according to the sliding strategy (past last row or to next row).

The semantics of the different match modes are explained by using an example.
Example: Ticker(Symbol, Tstamp, Price) is a relational table which stores historical stock data. Data in Ticker is displayed in table 4.3. Row # is not part of the schema, but it is added for ease of referring a record and for the remaining of this example, abbreviation r<num of row> is used as r1, which means first row.

Row #	Symbol	Tstamp	Price
1	A	1	90
2	B	2	100
3	B	3	150
4	B	4	30
5	A	5	98
6	B	6	51
7	A	7	100
8	B	8	176

Table 4.3: Ticker table

For convenience, we skip the irrelevant parts of the query and define the pattern as follows:

```
AFTER MATCH SKIP TO NEXT ROW
PATTERN (B*, D*)
   DEFINE B AS (B.Symbol = 'B')
          D AS (D.Symbol = 'D')
```

ALL match reports all the matches in the table based on the slide strategy. Therefore, match reporting feature is true other than ClosedFullMatch state. In this matching mode, since matches happened at OpenFullMatch state are reported, there is no new match at ClosedFullMatch state. Result of this query is: [r2], [r2, r3], [r2, r3, r4], [r3], [r3, r4], [r4], [r6]. Table 4.4 illustrates the execution of the above query on the table for matching mode ALL.

INCREMENTAL match evaluates the window in streaming fashion. So, it never goes back to the execution. In order to provide this feature, match reporting is set to false at ClosedFullMatch state, until new record is read. Result of the query is: [r2], [r2, r3], [r2, r3, r4], [r6]. Table 4.5 illustrates the execution of the query on the Ticker table for matching mode INCREMENTAL.

Window	Report Match?	Execution Status	Action	Match Result
[r1]	true	NonMatch	Slide by one	[ ]
[r2]	true	OpenFullMatch	Extend the window	[r2]
[r2, r3]	true	OpenFullMatch	Extend the window	[r2, r3]
[r2, r3, r4]	true	OpenFullMatch	Extend the window	[r2, r3, r4]
[r2, r3, r4, r5]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r3]	true	OpenFullMatch	Extend the window	[r3]
[r3, r4]	true	OpenFullMatch	Extend the window	[r3, r4]
[r3, r4, r5]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r4]	true	OpenFullMatch	Extend the window	[r4]
[r4, r5]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r5]	true	NonMatch	Slide by one	[ ]
[r6]	true	OpenFullMatch	Extend the window	[r6]
[r6, r7]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r7]	true	NonMatch	Slide by one	[ ]
[r8]	true	NonMatch	Slide by one	[ ]

Table 4.4: Execution of the example query for matching mode ALL

LOCAL MAXIMAL match reports the longest sequence of match in a window. Therefore, it reports the match at the ClosedFullMatch execution stage of each window which ensures that the match is not going to be longer. Result of the query is: [r2, r3, r4], [r3, r4], [r4], [r6]. Table 4.6 illustrates the execution of the same query on the Ticker table for matching mode LOCAL MAXIMAL.

MAXIMAL match reports the longest sequence of the distinct matches. Thus, it reports the matches only at ClosedFullMatch stage of the window evaluation. Furthermore, it does not report any matches during the sliding unless it produces a distinct match. This can happen when a pattern has some repetitive sub-patterns like pattern 'abcfgdab' where 'ab' is a common sub-pattern. This behavior is provided by setting the match reporting property true when a new record is read from the table. Result of the query is: [r2, r3, r4], [r6]. Table 4.7 illustrates the execution of the same query on the Ticker table for matching mode MAXIMAL.

Window	**Report Match?**	**Execution Status**	**Action**	**Match Result**
[r1]	true	NonMatch	Slide by one	[ ]
[r2]	true	OpenFullMatch	Extend the window	[r2]
[r2, r3]	true	OpenFullMatch	Extend the window	[r2, r3]
[r2, r3, r4]	true	OpenFullMatch	Extend the window	[r2, r3, r4]
[r2, r3, r4, r5]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r3, r4, r5]	false	NonMatch	Slide by one	[ ]
[r4, r5]	false	NonMatch	Slide by one	[ ]
[r5]	false	NonMatch	Slide by one	[ ]
[r6]	true	OpenFullMatch	Extend the window	[r6]
[r6, r7]	false	ClosedFullMatch	Slide according to the strategy	[ ]
[r7]	false	NonMatch	Slide by one	[ ]
[r8]	true	NonMatch	Slide by one	[ ]

Table 4.5: Execution of the example query for matching mode INCREMENTAL

4.2.1.3 FSM Runtime State

Execution_state structure stores FSM run time state. It is updated at each turn of FSM run. Main fields of the Execution_state are pointer to the current state, input buffer (see Section 4.2.3), last partial match added to the match buffer and execution status (see Section 4.2.1.2). Since multiple FSM instances may be running at the same time, it is necessary to differentiate the run time state of the FSM from its static state. This happens when FSM has to split because of non-determinism caused by non-mutual exclusivity of edge predicates or when a FSM instance is created with each new record. In such cases, there is no need to clone the whole FSM including state, edge information which are same for each FSM instance. Instead of it, only the run time of the FSM, which specified by Execution_state, is copied and added to FSM runs. In our current implementation, there is only the main run and in case of non-determinism, a precedence rule among the edges is applied. The rule is simple: take edge has priority over proceed edge. In another words, states are greedy and consume as much input as they can. There are other ways to handle the non-determinism and we are planning to explore this issue further.

Window	**Report Match?**	**Execution Status**	**Action**	**Match Result**
[r1]	false	NonMatch	Slide by one	[ ]
[r2]	false	OpenFullMatch	Extend the window	[ ]
[r2, r3]	false	OpenFullMatch	Extend the window	[ ]
[r2, r3, r4]	false	OpenFullMatch	Extend the window	[ ]
[r2, r3, r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[r2, r3, r4]
[r3]	false	OpenFullMatch	Extend the window	[ ]
[r3, r4]	false	OpenFullMatch	Extend the window	[ ]
[r3, r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[r3, r4]
[r4]	false	OpenFullMatch	Extend the window	[ ]
[r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[r4]
[r5]	false	NonMatch	Slide by one	[ ]
[r6]	false	OpenFullMatch	Extend the window	[ ]
[r6, r7]	true	ClosedFullMatch	Slide according to the strategy	[r6]
[r7]	false	NonMatch	Slide by one	[ ]
[r8]	false	NonMatch	Slide by one	[ ]

Table 4.6: Execution of the example query for matching mode LOCAL MAXIMAL

4.2.2 Match Buffer

An array of match buffers is used to report results. A match buffer is attached to each state and it stores partial matches called match tuples. Match buffer is actually a linked list of match tuples. Each match tuple has a pointer to the result record, a flag that shows whether it is the final tuple or not and a pointer to the previous match tuple. Match buffer provides a method to traverse the list. The list is traversed backwards during match reports.

4.2.3 Input Holder and Input Buffer

An input holder is used to store the active records of the table. A passive record means that there is no FSM runs on this record. Each pattern query has one input holder. The idea is to use input holder to store active records in order to use them during window sliding instead of asking for the table record, which might require disk access based on the storage engine. In this way, each record is read once. Another data structure which deals with input is input

Window	Report Match?	Execution Status	Action	Match Result
[r1]	true	NonMatch	Slide by one	[ ]
[r2]	true	OpenFullMatch	Extend the window	[ ]
[r2, r3]	true	OpenFullMatch	Extend the window	[ ]
[r2, r3, r4]	true	OpenFullMatch	Extend the window	[ ]
[r2, r3, r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[r2, r3, r4]
[r3]	true	OpenFullMatch	Extend the window	[ ]
[r3, r4]	true	OpenFullMatch	Extend the window	[ ]
[r3, r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[ ]
[r4]	true	OpenFullMatch	Extend the window	[ ]
[r4, r5]	true	ClosedFullMatch	Slide according to the strategy	[ ]
[r5]	true	NonMatch	Slide by one	[ ]
[r6]	true	OpenFullMatch	Extend the window	[ ]
[r6, r7]	true	ClosedFullMatch	Slide according to the strategy	[r6]
[r7]	true	NonMatch	Slide by one	[ ]
[r8]	true	NonMatch	Slide by one	[ ]

Table 4.7: Execution of the example query for matching mode MAXIMAL

buffer. An input buffer is used to handle sliding of a FSM run. In another words, input buffer is a subset of the input holder records and represents the active records of a run. As described in Section 3, after a match is found, the semantic window (FSM) slides according to the slide strategy. In case of non match, it slides by one in order to catch further possible matches. The input buffer keeps a pointer to the match tuple (which is the beginning of the window) and a pointer to the current match tuple. The input buffer is updated during the execution of a run based on execution status and it decides when it should read a new record from the input holder.

The reason for why we have two different buffers is due to the different usage of them. While match buffer is used at the end of an execution and stores result tuples on which projection, or aggregation on the records might be applied, input buffer is used at the beginning of an execution and stores the raw records.

Figure 4.1 shows the class diagram of our implementation.

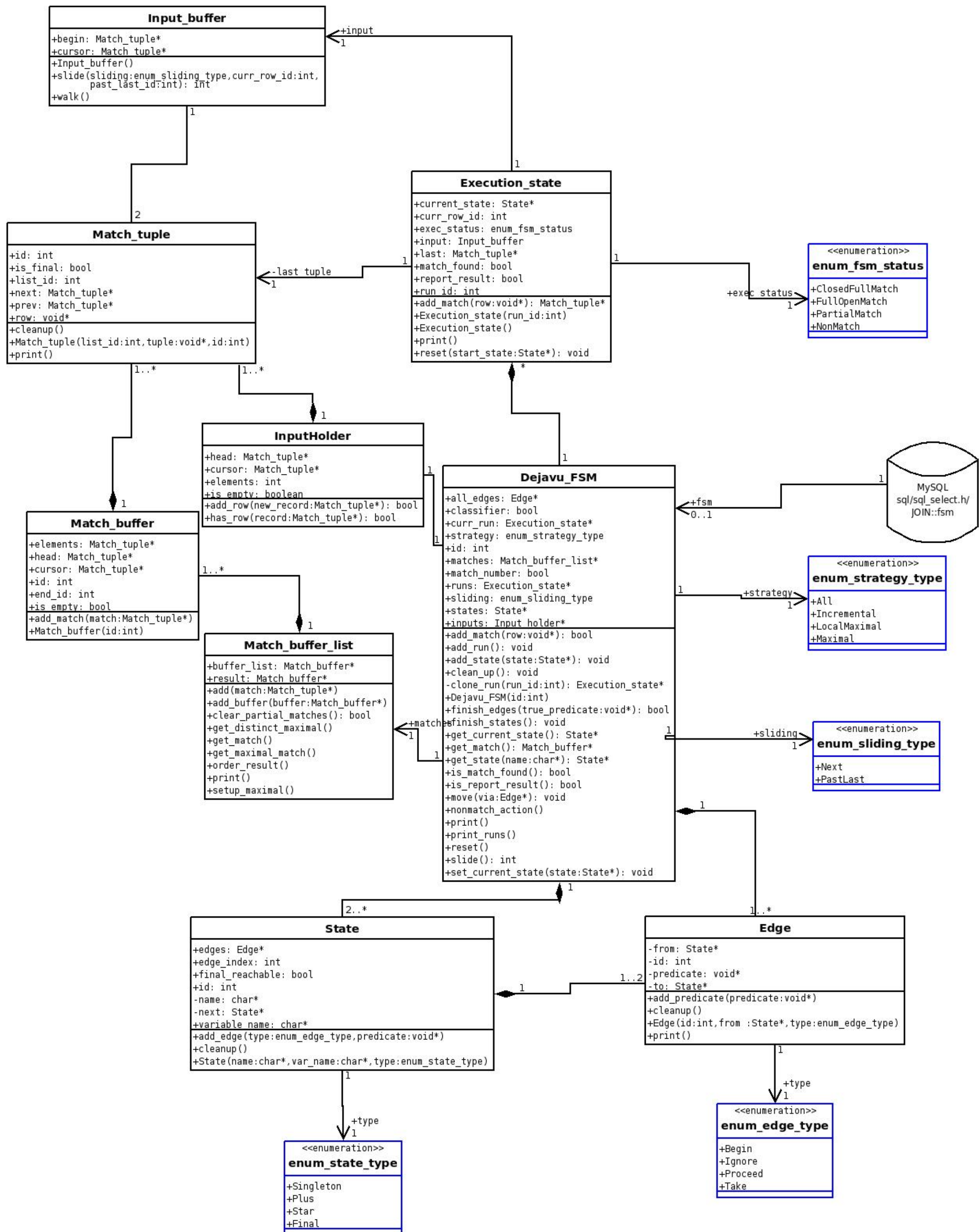

Figure 4.1: Class Diagram of MATCH-RECOGNIZE Implementation

4.3 Use Case

In this section, the goal is to explain the execution of an example query in our engine step by step as explained in 4.2.1. Let Stock(Symbol, Stock_date, Stock_time, Price, Volume) be an

append-only table with five columns representing historical stock prices. Row # is not part of the schema, but it is added for ease of reading. Some part of the data in Stock is shown in Table 4.8. The distribution of the whole dataset is showed in Figure 4.2.

Row #	Symbol	Stock_date	Stock_time	Price	Volume
...	...	...	...	...	...
37	A	2006-01-26	11:22:34	34.27	300
38	A	2006-01-26	11:22:42	34.26	100
39	A	2006-01-26	11:22:45	34.25	200
40	A	2006-01-26	11:22:52	34.27	900
41	A	2006-01-26	11:22:55	34.27	300
...	...	...	...	...	...
89	A	2006-01-26	11:30:06	34.23	100
90	A	2006-01-26	11:30:37	34.23	500
91	A	2006-01-26	11:30:42	34.21	100
92	A	2006-01-26	11:30:46	34.2	900
93	A	2006-01-26	11:30:47	34.22	300
94	A	2006-01-26	11:30:58	34.23	300
95	A	2006-01-26	11:30:59	34.24	400
96	A	2006-01-26	11:31:01	34.24	200
...	...	...	...	...	...

Table 4.8: Stock Table

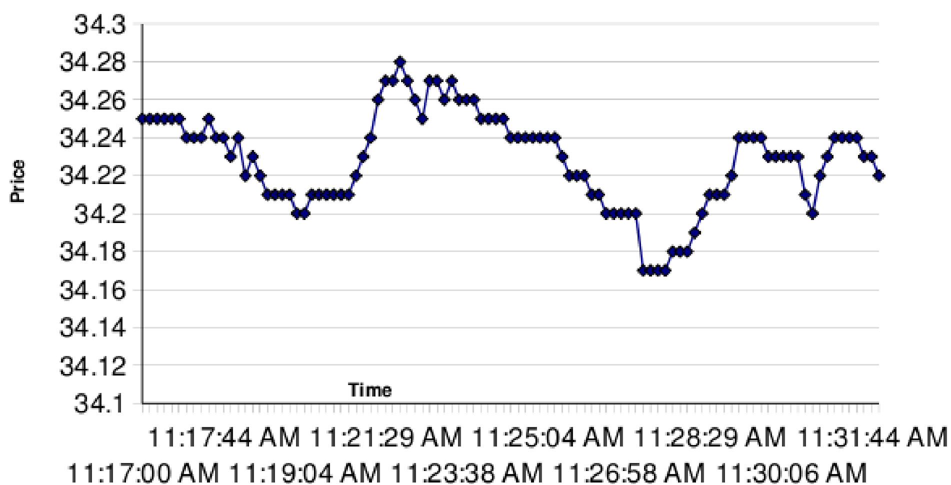

Figure 4.2: Time vs. Price Graph of Stock Table Data

The example query detects the following pattern in price: a falling price, followed by a rise in price goes higher than the price was when the fall began.

```
SELECT *
FROM Stock MATCH_RECOGNIZE (
   MEASURES
      MATCH_NUMBER AS Matchno,
      CLASSIFIER AS Classy
   AFTER MATCH SKIP TO NEXT ROW
   ALL MATCH
   PATTERN(A B+ C* D)
      DEFINE B AS (B.Price < A.Price AND B.Price <= PREV(B.Price))
             C AS (C.Price > PREV(C.Price) AND C.Price <= A.Price)
             D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
);
```

The grammar rules module of the MySQL parses the query according to the rules defined in sql/sql_yacc.yy. Grammar file initializes the parse tree structure and creates an FSM as explained in 4.2.1.1. The following figure shows the corresponding Finite State Machine of the query.

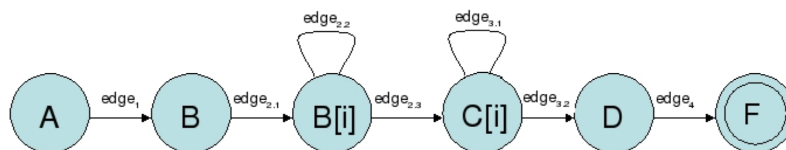

Figure 4.3: FSM

name	predicate
$edge_1$	true
$edge_{2.1}$	B.Price < A.Price & B.Price <= PREV(B.Price)
$edge_{2.2}$	B.Price < A.Price & B.Price <= PREV(B.Price)
$edge_{2.3}$	true
$edge_{3.1}$	true
$edge_{3.2}$	C.Price > PREV(C.Price) AND C.Price <= A.Price)
$edge_4$	D.Price > PREV(D.Price) AND D.Price > A.Price

Table 4.9: Predicates of FSM edges

FSM is a member of the MySQL JOIN class. Every SELECT query in MySQL is considered a join. Join simply stores the major information of the select query like where expression tree, table descriptors of the join. Afterwards, it leads to the execution of mysql_pattern(...) from sql/sql_select.cc. The relevant code snippet for mysql_pattern method looks as follows:

```
if (err= join->prepare(rref_pointer_array, tables, wild_num,
                           conds, og_num, order, group, having, fsm, proc_param,
                           select_lex, unit))
  {
    goto err;
  }
  if (join->flatten_subqueries())
  {
    err= 1;
    thd->net.report_error= 1;
    goto err;
  }
  if ((err= join->optimize()))
  {
    goto err; // 1
  }
  if (thd->net.report_error)
    goto err;
  join->exec_pattern();
  if (thd->cursor && thd->cursor->is_open())
  {
    /*
      A cursor was opened for the last sweep in exec().
      We are here only if this is mysql_select for top-level SELECT_LEX_UNIT
      and there were no error.
    */
    free_join= 0;
  }
 ....
```

At the preparation phase of the join, edges of the FSM is initialized to be able to run them on records later. join::flatten_subqueries() converts candidate subquery predicates to semi-joins. Since we have not deal with nested queries yet, this part is skipped. As will be explained in Chapter 5, optimizer of MySQL does not treat pattern queries differently. Finally, we reach

join::exec_pattern which is responsible for the execution of the pattern queries. Exec_pattern execution is divided into two: execution of constant and non-constant tables. A table is considered constant if it contains at most one record, or if at most one record match is possible during a key lookup (the key is either primary or unique). Execution of pattern queries follows non-constant table execution branch. Then execution leads to sub_select method in sql_select.cc which asks for the record, retrieves matching full record, and sends them to the result set stream. More specifically, sub_select method reads record with read_record method of JOIN_TAB read record descriptor. As a reminder, the data access method was defined at optimization phase, for pattern queries the method is simple table scan. Afterwards, it evaluates the record with evaluate_join_record method in the sql_select.cc class. If the record satisfies the condition, evaluate_join_record calls end_send method to send the record to the result set stream. sub_select is the place where logic explained in 4.2.1.2 is implemented and records are added to the input buffer. FSM runs inside the evaluate_join_record method. When the proceed edge is taken, evaluate_join_record is called again since proceed edge does not consume the input. So, evaluate_join_record is the place where edge logic is applied. In addition to that, the state of the FSM is also updated here. Furthermore, when a predicate on consume edge is satisfied, Match_tuple is created from the record and Match_tuple is added to the Match_buffer. In the end_send method, if match is found, result is reported by traversing the match buffer. The execution of the sample query in Stock table is based on the logic explained in section 4.2.1.2, but it is important to mention about the match buffer and input buffer of the execution. The following figure shows the status of match and input buffer when last matches are detected.

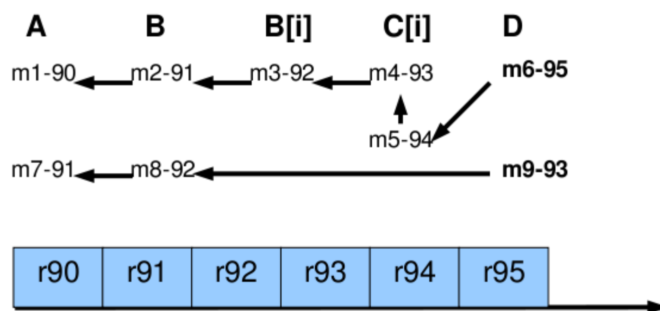

Figure 4.4: FSM Execution

A, B, B[i], C[i] and D in Figure 4.4 are the states. A match buffer is attached to each state. m1-90 is a match tuple with id 1, which has pointer to the row 90. As stated before, match tuple encapsulates the row information and attaches metadata to it. The big arrow in the figure represents the input buffer which holds the active records and the records inside the boxes (colored blue) are part of the semantic window. The bold tuples are final tuples of the

match. So result is reported by following previous pointers of the final match tuple. Result of the query is as follows:

Symbol	**Stock_date**	**Stock_time**	**Price**	**Volume**	**Matchno**	**Classy**
A	2006-01-26	11:22:42	34.26	100	0	A
A	2006-01-26	11:22:45	34.25	200	0	B
A	2006-01-26	11:22:52	34.27	900	0	D
A	2006-01-26	11:30:37	34.23	500	1	A
A	2006-01-26	11:30:42	34.21	100	1	B
A	2006-01-26	11:30:46	34.2	900	1	B[i]
A	2006-01-26	11:30:47	34.22	300	1	C[i]
A	2006-01-26	11:30:58	34.23	300	1	C[i]
A	2006-01-26	11:30:59	34.24	400	1	D
A	2006-01-26	11:30:42	34.21	100	2	A
A	2006-01-26	11:30:46	34.2	900	2	B
A	2006-01-26	11:30:47	34.22	300	2	D

Table 4.10: Query result for sliding strategy next row

Result of the same query for sliding strategy past last row is showed in Table 4.11. Since past last row strategy ensures non-overlapping matches, the overlapping match result [r91, r92, r93] is skipped.

Symbol	**Stock_date**	**Stock_time**	**Price**	**Volume**	**Matchno**	**Classy**
A	2006-01-26	11:22:42	34.26	100	0	A
A	2006-01-26	11:22:45	34.25	200	0	B
A	2006-01-26	11:22:52	34.27	900	0	D
A	2006-01-26	11:30:37	34.23	500	1	A
A	2006-01-26	11:30:42	34.21	100	1	B
A	2006-01-26	11:30:46	34.2	900	1	B[i]
A	2006-01-26	11:30:47	34.22	300	1	C[i]
A	2006-01-26	11:30:58	34.23	300	1	C[i]
A	2006-01-26	11:30:59	34.24	400	1	D

Table 4.11: Query result for sliding strategy past last row

In order to prove the correctness of the result, the distribution of data is displayed where matches are highlighted and displayed in a circle in Figure 4.5. In the figure, points inside a

circle are (colored different than blue) part of the match. It is seen that there is a circle inside a bigger circle (green points are inside the red points), where overlapping matches appear. Therefore, in this case both of these circles are matches, where bigger circle covers the points of the small circle (red points constitute a match including the green points and green points are another match).

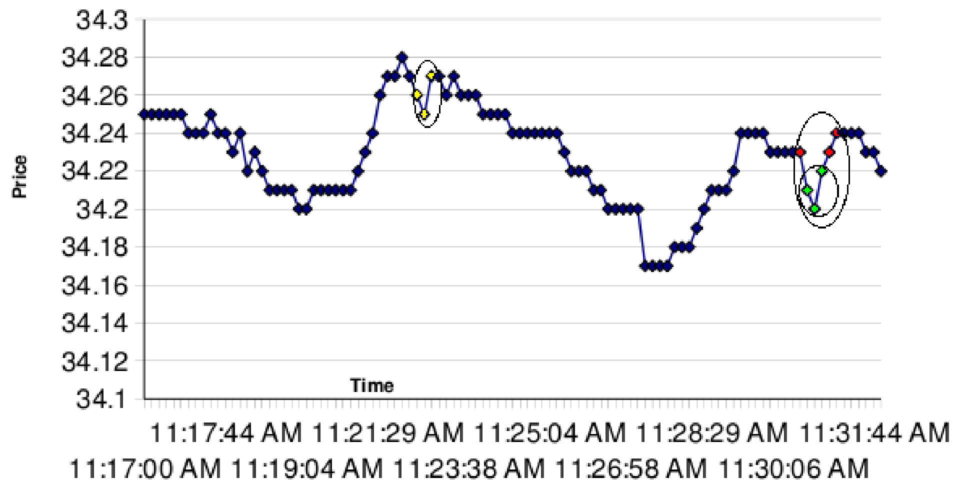

Figure 4.5: Query result on data graph

After the result is sent to the user, cleaning of select structures is performed. For pattern queries, FSM machine is cleaned here including all its edges, states, match tuples, and input holder and buffer.

4.4 Statistics

Table 4.12 gives information about source lines of the code of MATCH-RECOGNIZE implementation. The changes are grouped into two parts: inside the MySQL and outside the MySQL. For the source code added to the MySQL, the extensions are classified per module. During the implementation, we tried to reuse existing MySQL architecture and modules as much as possible while minimizing our extensions. Although lines of code can not be considered as a precise proof of this effort, it gives at least an idea about it. MySQL Core has almost ∼360,000 lines of code. With the word core, we mean the source code which is responsible for parsing and execution of the query. Thus, parts which provide platform independence, BUILD packages, and storage functionality are excluded. As it can be seen from the table, most of the code (58%) is implemented outside MySQL than inside MySQL. For the code added to MySQL, big part of it is added to the Parser module. The Parser module encapsulates both lexer and grammar rule file. Even the codes implemented inside the MySQL Parser is less than 7% of the whole Parser

Category	Module	Extension (SLOC)	Original (SLOC)
Inside MySQL	Parser	~800	~12000
	Connection & Initialization	~110	several thousands
	Execution	~440	~18000
Outside MySQL	FSM	~1280	-
	Match_buffer	~540	-

Table 4.12: Source Lines of Code

codes. Similarly, the extensions performed to the SELECT query module is less than 3% of itself. Furthermore, approximately 110 lines of code is added for connection and initialization. Finally the total lines of codes added to the MySQL core corresponds less than 0.4% of itself, while number of total lines of MATCH-RECOGNIZE code corresponds less than 1% of MySQL core's line of codes.

Chapter 5

Architecture

MySQL is an open source relational database management system (RDBMS) which was first released in May of 1996 [9] and widely used all around the world. For our research project we choose to extend MySQL, because it is highly accepted in open source community and offers a comprehensive range of documents and consulting support like mysql internal mailing list, which turned out to be very helpful. In addition to these, MySQL's pluggable storage engine architecture gives the flexibility to choose among existing storage engines based on the application per table. It also gives opportunity to develop customized storage engines to optimize the engine for specific purposes by using a well defined, clean interface. Currently, we have not touched the data access optimizations, but it is one of the issues that we want to investigate as future work (discussed in Chapter 6).
The goal of this chapter is to explain the role and effect of our pattern module on the MySQL architecture. The remainder of the chapter explains the original MySQL architecture, and our pattern matching module extensions to this architecture.

5.1 MySQL Architecture

The content of this section is heavily based on the book "Understanding MySQL Internals"[9]. In order to understand the MySQL architecture, it is worth to mention briefly about MySQL source code's history.
MySQL has even some code pieces which are more than 20 years old, which were written by Monty before it is called MySQL. When most of the code was originally written, it was not done to be part of some big system, instead to solve very specific problems [9]. Because of it, it is very difficult to talk about a well defined, modular architecture, but there is a certain high level functionality and code mappings. While explaining these mappings, we will also use the term "module" as Sasha Pachev, author of "Understanding MySQL Internals". Before going further with the details of the MySQL architecture, we would like to clarify definition of term "module" in MySQL architecture context. The term *module* is used as a piece of code that logically belongs

together in some way and performs a certain critical function in the server, instead of its usual meaning independent code piece which you can be replace with another implementation easily [9]. See Figure 5.1 for general view of the core modules and their interactions. The modules having stars are the modules that we have extended for pattern matching. The extensions are discussed in Section 5.2.

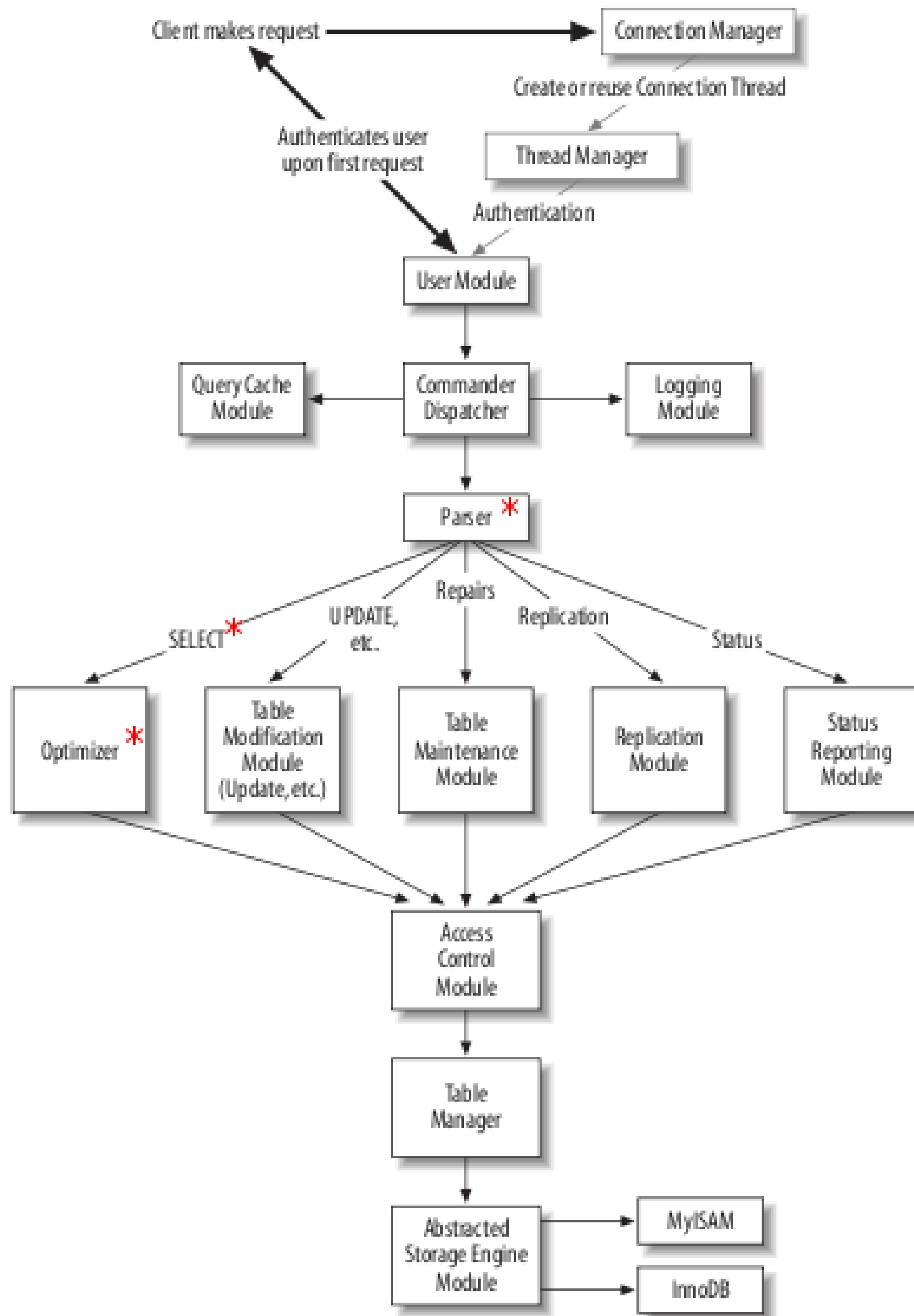

Figure 5.1: High-level view of MySQL modules[9]

5.1.1 Interaction of the Core Modules

In order to understand the MySQL architecture better, let us explain the execution of a query. Whenever MySQL Server is started, the Server Initialization Module sets up the server by initializing global variables and structures, allocating global memory buffers, and performing some other start-up tasks. After the initialization is done, it passes the control to the Connection Manager. Connection Manager handles the communication protocol tasks and waits for the client requests. Whenever it receives a request from a client, Thread Manager module creates a new thread or retrieves a thread from the cache to take care of the request. The Connection Thread, which is responsible for the request, first connects to the User Module to check the authentication of the user. If the user is verified, the request is forwarded to the Commander Dispatcher. Based on logging mode, Commander Dispatcher asks Logging Module to log the query before processing it. Afterwards, the Commander Dispatcher forwards the queries to the Parser through the Query Cache Module. If the query is cached and there exists a stored result which is still valid, result is sent to the user. This way the Connection Thread becomes ready to process another command. In case of a cache miss, the query goes to the Parser. While parsing the query, parser also creates the internal structure of the query and forwards the query to different modules based on its type. As it can be seen from Figure 5.1, SELECT queries are forwarded to the Optimizer; updates, inserts, deletes, table creation and table alters are forwarded to the Table Modification Module; queries that maintain tables like back up, repair, restore are forwarded to the Table Maintenance Module; queries related to the replication are sent to Replication Module; and queries about server configuration, table structure information, and so on are sent to the Status Reporting Module. We will concentrate on the execution of the SELECT query which is forwarded to Optimizer module. Though its name is Optimizer, Optimizer module does more than this. This module is responsible for the whole execution of SELECT queries. Each of these modules need to access a list of tables which are passed through the Parser module. In this case, they connect to the Access Control Module. Access Control Module checks whether the user have required rights to access the data needed for the query. If the user has the required privileges, the Table Manager Module opens the table and obtains required keys. From now on, storage engine can be used to perform low level data tasks like insert, update, fetch data. For this purpose, control is passed to the Abstracted Storage Engine Module. As stated before, MySQL supports pluggable storage engine architecture. Abstracted Storage Engine Module calls specified storage engine by using polymorphism. Therefore, this module can be considered as an interface between MySQL and other storage engines. While query is processed, result might be sent to the user. Once the query is done, the control passes to the Connection Thread again. Connection Thread makes the cleanup and waits for the further queries from the user until user quits. The execution flow of non-regular client requests like replication slave is skipped, since these are beyond the scope of this thesis.

5.2 Pattern Matching Module Extensions

This section discusses the changes that we made in the MySQL engine to extend it with pattern matching capability. We tried to make as fewer changes as possible to MySQL, and used every opportunity to reuse the existing MySQL constructs. Remainder of this section presents our extensions to each MySQL module in detail.

5.2.1 Parser

MySQL has a hard-coded lexer, but uses a Bison parser to define the grammar rules. The MATCH-RECOGNIZE keywords are added to the lexer file of MySQL (sql/lex.h) and the yacc file of MySQL (sql/sql_yacc.yy) is extended to encapsulate the parsing rules of MATCH-RECOGNIZE clause (see appendix A for EBNF of the MATCH-RECOGNIZE grammar). The parser module has two important roles: checking the syntax and creation of the internal representation of the query. Let us start with the grammar rules we added, later we will mention about the construction of FSM while applying these rules.
The first connection point between Match Recognize and MySQL rules is the table specification at the FROM rule with 'match-recognize' rule as it can be seen from the snipped code of MySQL grammar file.

```
derived_table_list:
          table_ref
          {
            LEX *lex= Lex;
            lex->sql_command= SLOCUM_SELECT;
            $$=$1;
          }
        | table_ref match_recognize_clause
          {
            LEX *lex= Lex;
            lex->sql_command= SLOCUM_PATTERN;
            $$=$1;
          }
        | derived_table_list ',' table_ref
          {
            MUSCLE_ABORT_UNLESS($1 && ($$=$3));
          };
```

For the rest of parsing, MATCH-RECOGNIZE connects back to the MySQL rules while parsing the expressions at DEFINE clause. As you might remember, DEFINE syntax is like

"DEFINE A AS expr, B AS expr" etc. Since regular relational predicates can be used in DEFINE clause, we decided to use MySQL expr here. MySQL expr is the SQ expression which is used in WHERE, HAVING, GROUP BY, ORDER BY, or the field list of a select query. However, DEFINE clause requires more than the regular predicates of SQ. Therefore, it was necessary to extend the MySQL expression rules in order to handle pattern variable correlation like "A.Price" and "PREV(A.Price)" type of field access which can be used at DEFINE clause. By modifying the expr rule, which is widely used by MySQL parser, we didn't forget to add control statements in order to not break the original MySQL functionality. Another remark about the parser is that re-use of select alias rule while parsing MATCH_NUMBER and CLASSIFIER clauses. This was the obvious choice, since they have very similar syntax. As a reminder, MATCH_NUMBER syntax is like following: "MATCH_NUMBER AS mach_no". CLASSIFIER also has the same syntax.

As we stated before, Parser is also responsible for creating the internal representation of the query. Therefore, FSM creation and initialization are also performed during parsing (which were discussed in detail in Section 4.2.1.1). Here, we would like to concentrate more on extensions to the MySQL architecture.

We reused MySQL expressions in DEFINE clause, since it is not very different than WHERE clause in SQ. While parsing a predicate, MySQL Parser creates the Item tree and at the execution phase, each node in the item tree is recursively evaluated in order to find out whether the given predicate is satisfied by the record. The Item class defined in sql/item.h is the base class for all other Item_* classes, which represents the nodes of an expression tree. This family of classes covers arithmetic operations, various SQ functions, logical operators such as AND and OR, references to the table fields, subqueries returning one row etc [9]. Field correlation and PREV operator were not part of the SQ expression, so we introduced two new Item types, in another words two new nodes to the parse tree: Item_match_field and Item_prev_field. Both classes extend from Item_field, which is field access node of expression tree. Different than Item_field, both Item_match_field and Item_prev_field store variable name and field variable name. Variable name is the name of the variable which Item_match_field and Item_prev_field nodes belong. On the other hand, field_varname represents the name of the variable whose field needs to be accessed. The parse tree of the definition of D variable in example query of Section 3.1 is shown in Figure 5.2. In the example, while all Item_match_field and Item_prev_field nodes have the same variable name 'D', their field variable names are different as shown in Figure.

```
...
PATTERN(A, B+, C+, D)
DEFINE ...
       D AS (D.Price > PREV(D.Price) AND D.Price > A.Price)
```

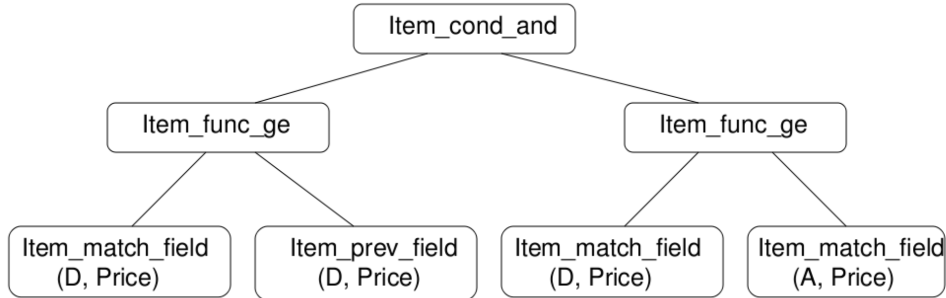

Figure 5.2: Parse Tree of Pattern Variable D

Moreover, Item_match_field and Item_prev_field have also pointer to the FSM instance in order to reach the correct Match_duple during the execution of the item tree. For example, match_field item uses field variable name as a key to search the Match_duple inside the match buffer through the FSM. The search and binding of Item with records has to be done during the execution, since variable names are attached to the rows during the execution based on FSM state. On the other hand, in Item_field objects of MySQL, the structure is filled with the record data when the record is read. When we go back to field correlation items, match_field and prev_field has different behaviour. While match_field returns the field of the found duple in match buffer, prev_field returns the field of the row stored before the found duple in the match buffer.

Another main constructs we added to the Parser module are MATCH_NUMBER and CLASSIFIER. The effect of MATCH_NUMBER and CLASSIFIER is similar to the having constant field in selection list like "SELECT 0 as Zero FROM table" and/or having aggregation in selection list like "SELECT Count(*) FROM table". It is similar to the first one, since the "Zero" column is added to each resulting row, on the other hand it is similar to the second one since the value of the column is determined during execution. The first one is handled in MySQL by creating a constant item field at Parser based on type of the column. For the example above, Item_int would be created. For the second one Item_sum_count object is created which inherited from Item_sum class. Item_sum is a base class for special SQL expressions called set expressions like MAX, AVG, COUNT etc. In case of MATCH_NUMBER, Item_match_num_field instance is created during parsing. Item_match_num_field extends from Item_int, but its value can be changed during execution like Item_sum instances. With the same approach, Item_classifier extends from Item_string. It has similar functionality as Item_match_num_field. The Item_classifier is created when CLASSIFIER is requested in the query. At the execution of the FSM (see 5.2.3), the value of both CLASSIFIER and MATCH_NUMBER are updated if necessary based on FSM status.

5.2.2 Select

After the parse tree is initialized, execution of SELECT queries without MATCH-RECOGNIZE leads to the handle_select() method in sql/sql_select.cc. Inside handle_select() method, the queries which require unions of tables are forwarded in mysql_union() in sql/sql_union.cc, while the single select queries are forwarded to mysql_select() method at sql/sql_select.cc. Differently, for the queries with MATCH-RECOGNIZE defined as Pattern command, execution leads to the handle_pattern() method and mysql_pattern() method in sql/sql_select.cc afterwards. In this way, pattern queries are considered special type of the select queries. With the mysql_pattern method besides all the required information for a simple SELECT query like list of fields, tables; the FSM created during the parsing of Match_Recognize is also passed to the Optimizer module.

5.2.3 Optimizer

In MySQL context, Optimizer refers to the server module responsible for creating and executing the plan to retrieve the records requested by the query as soon as possible. Therefore, the optimizer picks the order in which the tables are joined, the method to read the records (e.g., read from an index or scan the table), as well as which keys to use [9]. In addition to the decision making, Optimizer also executes the query and does cleanup afterwards. JOIN is a structure to represent SELECT query in MySQL. Even a query only with one table is considered as JOIN. Execution of a single SELECT consists of the following steps:

1. JOIN::prepare*( )
2. JOIN::optimize( )
3. JOIN::exec*( )
4. JOIN::cleanup*( )

Methods with * are the methods which are extended for pattern query execution.

5.2.3.1 JOIN::Prepare()

JOIN::prepare() performs a series of initializations, the following being the most important ones:

- Prepare the tables and check access for the view tables. Table preparation also includes adding of additional information to the tables which are going to be used for optimization purposes later.
- Replace wild cards (*) in fields with actual names of the fields and initialize the field list descriptions.
- Initialize the field descriptions.

- Set up the WHERE, HAVING, and ON clauses.
- Set up ORDER BY and GROUP BY clauses.

In addition to these, setup of the subselects is also done in this phase. As an extension to the current structure, for pattern queries, the FSM is also initialized here and the conditions on the edges which are expressions on DEFINE clauses are set up here.

5.2.3.2 JOIN::Optimize()

JOIN::Optimize() is responsible for restructuring the query and creating an execution plan, including:

- Reconstructing the where and having clauses and eliminating any redundancies.
- Determining the access methods and keys of the tables based on query predicates and indexes defined on the tables.
- Determining the best join order based on queries and statistics of tables.

We have not yet done any optimization on pattern queries. It is one of the top items in our list of future work. Currently, MySQL optimizer is not aware of MATCH-RECOGNIZE structure, since it appears in the FROM part. For optimizer, the only interesting optimization performed based on FROM clause information is determination of the join order. The obvious opportunity is to use optimization of conditions for DEFINE clause which are same as WHERE clause. Another opportunity is to determine the best access method for the pattern queries. Currently, Optimizer picks simple "table scan" as a data access method for MATCH-RECOGNIZE queries, since MATCH-RECOGNIZE query is nothing more than a SELECT * FROM query from MySQL point of view.

5.2.3.3 JOIN::Exec()

JOIN::exec() is responsible for performing actual execution of the query. This is the main part we mostly changed. Changes can be grouped in the following way: Record Management, Predicate Execution, and Output Management. Each of them will be discussed in detail next.

Record Management: As stated before, JOIN structure is the key structure to store query plan of the SELECT queries. JOIN has an array of JOIN_TAB descriptors each of which contains the information relevant to the optimization about each table instance participating in a join. JOIN_TAB has data member read_record, which is a record reading descriptor. While running the queries, the next record is asked from read_record. Based on the access method decision made by the optimizer, read_record retrieves the next record either from a table data or an index file. Afterwards, the record is evaluated and the next record is asked until end of

records is reached. In case of a pattern clause, we do not ask for a new record at each execution iteration. As discussed in Chapter 4, FSM might process a record more than once based on its slide strategy. During sliding, after all the records stored at input holder are processed, we need to feed the FSM with a next record from table. In order to scan the table just once, we added an additional control mechanism on top of data access point of Optimizer, which is the sub_select() method in sql/sql_select.cc as described in Chapter 4.

Predicate Execution: In MySQL, evaluate_join_record() method in sql/sql_select.cc is a place that parts of the WHERE clause which can be applied to the partial records are evaluated. Originally, this method evaluates the same predicate on the current record. In case of pattern queries, predicates are attached to the edges of the FSM, and based on current state of FSM, edges of this state are evaluated on the record. As we discussed in Chapter 4, when proceed edge is taken (which does not consume the input) current state information of FSM is changed and the edges of the new current state's edges are evaluated on the record. In this case, many predicates are evaluated on a single record. Originally the same predicate is evaluated on each record in MySQL evaluate_join_record method. Therefore, we added our FSM execution here in order to provide the iteration of predicates based on current state information of FSM.

Another difference is that the method evaluates the predicate on the current record. However for pattern queries, we may need to run the predicates on a different record than the current one. For example, we need this capability when FSM evaluates the records inside the input buffer during window sliding. In addition to the iteration of predicates, we also needed to support the iteration of the records. Therefore, we added the 'move field' functionality. In conclusion, in order to be able to run pattern clauses we changed the evaluation of predicate mechanism to evaluation of states mechanism for pattern queries.

Output Management: In MySQL, the evaluation of one nested loop iteration ends at end_send() method in sql/sql_select.cc. If the result of the query is going to be sent to the client, this is the place where the resulting record is sent to the buffer after checking the existence of the HAVING condition. For pattern clauses, result is a set of rows and is ready when FSM reaches its final state. In case of pattern queries, we check if a match is found in the same way as checking the HAVING clause. Another important difference between pattern and non-pattern queries is the storage of partial matches. In case of pattern queries, partial matches have to be stored in a match buffer. In order to store a partial match, the current record should be copied. MySQL creates temporary tables for different purposes such as storing the intermediate results and performing too complicated sorts on data. We followed a similar approach in copying the partial results. MySQL makes deep copy of the fields in these cases. On the other hand, in pattern matching queries, only *ptr members of fields are copied, which point to the fields' data in the in-memory copy of the record. The reason for the difference is the following: it is enough to store only ptr value for pattern queries since copying is performed at the end of the execution. However MySQL needs deep copy at the early stages of execution phase and makes

deep copy of whole field structure.

5.2.3.4 JOIN::Cleanup()

Join::Cleanup() method frees up the resources of a given join, like unlocking tables and deleting Item trees. As an extension to the original cleanup method, we also need to free the FSM resources in case of pattern queries.

Before concluding the section about MySQL extensions, we would like to clarify an important point: Data Access. We haven't done any work related to the data access of the MATCH-RECOGNIZE queries yet. We would like to work on it as a future work item. Currently, pattern queries can run on any available storage engine (like any other query).

5.3 Summary

We will conclude the discussion about architecture by summarizing the extensions as shown in Table 5.3. We believe that similar extensions would also apply to other relational engines, when pattern matching functionality is to be added.

Functionality	**Discussion**	**Extensions**
CLASSIFIER, MATCH_NUMBER	MySQL: SELECT COUNT(*) AS & SELECT 0 AS zero Needed: one column per record & value set at execution	Re-use select alias New node: extends from constant node, but value can be set at execution
Field Correlation A.Price & PREV(A.Price)	MySQL: field access Needed: variable name + field variable name + execution depends on FSM	Re-use field access syntax New node: extends from field_item but with variable name and FSM based execution
Input mechanism	MySQL: -	Add control mechanism at MySQL data access point to ask for the row if required
Record execution	MySQL: one record, one predicate Needed: state execution	FSM states are the basis of the evaluation, one record 1..* predicates move field capability
Storing partial matches	MySQL: complex queries creates temporary table and copies the fields deeply	Store only pointers to the data of the fields
Output mechanism	MySQL: similar to HAVING clause on column	HAVING clause on whether FSM reached final state

Table 5.1: Summary of Pattern Matching Extensions on MySQL

Chapter 6

Conclusions & Future Work

The goal of the this was to extend the MySQL relational database systems to be able to support pattern matching queries on stored data sequences, where patterns are defined as regular expressions. In order to achieve this, the existing pattern languages are investigated and a major subset of the 2007 ANSI standard proposal [2] for adding MATCH-RECOGNIZE clause to standard SQL is implemented. The MATCH-RECOGNIZE clause is modelled as a semantic window and is internally represented as a Finite State Machine. In order to make the query run, we extended the Parser of the MySQL and created the FSM at the end of the parsing phase. In the execution phase, we replaced the read_record() mechanism of MySQL with the Input Holder of FSM so that the record reading is performed based on the window execution. We also replaced the static predicate-record evaluation mechanism of MySQL with dynamic predicate-record evaluation where the predicate is based on the state of the FSM and the record is based on state of the execution of FSM run. Furthermore, we also changed the minimum unit of output mechanism of MySQL for pattern queries to a match which is a set of rows. On the other hand, we made these changes in a modular way without breaking the current functionality of MySQL. We heavily reused the parser of MySQL and added MATCH-RECOGNIZE rules as a block to it. In addition to that, we followed the main module structure of MySQL.

The most challenging part of this thesis was to deal with the huge unknown MySQL source code. The MySQL core has roughly 360,000 lines of C/C++ source code. In such a big code it was more difficult to identify the correct place where to put the changes rather than to define the changes that are needed. For example, the source file of optimizer (sql_select.cc) that changed the most has around 18000 lines of code. During the implementation, the approach to find a mechanism in MySQL that is similar to the wanted functionality is used very frequently. Studying the similar mechanism was very helpful to find the correct place as well as to re-use the MySQL code. In the end, I believe that I gained valuable experience in dealing with a large code base.

Before concluding this thesis, we would like to mention about the limitations of our system

and future works. The main limitation of our system is lack of multiple FSM runs. Currently we can only run one FSM instance although the difference between static and dynamic information of the FSM is introduced. One FSM instance is a performance bottleneck and precedence rule between edges might cause the loose of the matches. Support of the multiple FSM runs is the main functionality which we want to add in near feature. In addition to that, to be able to pipeline or parallelize the multiple FSM instances are other ideas we want to investigate more in order to address performance bottleneck of multiple FSM runs. Another limitation of our system is that it looks only for strictly contiguous patterns. Currently, we are not able to handle PARTITION BY clause or provide the functionality to skip the irrelevant tuples. We are planning to work on support of non-contiguous patterns as future work. Introduction of non-contiguous pattern search gives the opportunity for different data access methods. Currently, the data access method for pattern queries is table-scan. It is a good option when patterns are contiguous, but with the PARTITION BY clause, where contiguity feature is relaxed, it is an option to use index or different access methods. In particular, when we have multiple FSM instances, it will become crucial to deal with the inputs and sliding of these FSMs. Therefore, input management of pattern clauses is another issue we want to concentrate on in the near future. Lastly, our work concentrates on finding patterns on sequence data. This data has an order as the word sequence implies. For this purpose, we are making the assumption that the data is ordered by time. The ordering is guaranteed by MySQL when the table is append only.

Besides removing the limitations of the system, we are also planning to work on issues which will improve the performance of our system or usability and development time. Post-match management is an idea to improve the performance of the system. Currently, after the pattern is found, the resulting matches are not stored in the system. Actually there is an opportunity to use the resulting matches to gather the statistics and/or feedback in order to make optimization decisions. Another idea is to store the match result in order to reuse in later pattern searches. On the other hand, FSM Visualizer is another future work which address the difficulty of understanding the system. FSM execution is complex and it is not easy to prove whether the system behaves as it should. Therefore, working on a real-time visualiser of FSM is a must in order to test and verify the functionalities as well as reducing the development time of the new features of system.

Acknowledgments

I want to conclude with saying thank you to Prof. Nesime Tatbul for guiding me through this thesis with big patience and understanding. I am deeply grateful to Roozbeh Derakhshan, Dr. Jens Dittrich and Irina Botan who gave useful suggestions and ideas about my thesis design and implementation. I would like to thank Dr. Peter Fischer for his technical support whenever I needed.

Amina Fazlic and Nida Farid receive my thanks for helping me with English corrections.

Last but not least, I am very thankful to my family for having supported me morally and financially and having made my studies possible.

Appendix A

EBNF of MATCH_RECOGNIZE

The rules written as < ... > are the native MySQL parsing rules.

```
derived_table_list:      <table_ref> [match_recognize_clause]
                         | derived_table_list ',' <table_ref>

match_recognize_clause: MATCH_RECOGNIZE '(' [measures_clause] pattern_clause ')'

measures_clause:         MEASURE [measure_options]

measure_options:         [match_num] [classifier]

classifier:              CLASSIFIER <select_alias>

match_num:               MATCH_NUMBER <select_alias>

future_strategy:         AFTER MATCH SKIP match_qualifier

match_qualifier:         TO NEXT ROW
                         | PAST LAST ROW

selection_strategy:      LOCAL_MAXIMAL MATCH
                         | ALL MATCH
                         | MAXIMAL MATCH
                         | INCREMENTAL MATCH
```

```
pattern_clause:          [future_strategy] [selection_strategy]
                         PATTERN '(' reg_term_list ')'
                         DEFINE pattern_def_list

reg_term_list:           reg_term
                         | reg_term reg_term_list

reg_term:                IDENT [ operator ]

operator:                * | +

pattern_def_list:        pattern_def
                         |  pattern_def pattern_def_list

pattern_def:             IDENT AS '(' <expr> ')'
```

Bibliography

[1] Coral8. In *http://www.coral8.com.*

[2] Pattern Matching in Sequences of Rows. In *http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf*, March 2007.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient Pattern Matching over Event Streams. In *SIGMOD Conference*, Vancouver, Canada, June 2008.

[4] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae: History-Enhanced Monitoring. In *CIDR Conference*, Asilomar, California, January 2007.

[5] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subcribe Systems. In *EDBT Conference*, Munich, March 2006.

[6] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR Conference*, Asilomar, California, January 2007.

[7] Y. Diao, N. Immerman, and D. Gyllstrom. SASE+: An Agile Language for Kleene Closure over Event Streams. In *Technical Report 07-03*, UMass Amherst, 2007.

[8] S. McReynolds. Complex Event Processing in the Real World. In *An Oracle White Paper*, www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf, September 2007.

[9] S. Pachev. *MySQL Internals.* O'Reilly, 2007.

[10] S. Rizvi. Complex Event Processing Beyond Active Databases: Streams and Uncertainties. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2005.

[11] J. M. Robert D. Edwards. *Technical Analysis of Stock Trends.* AMACOM, 1997.

[12] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Optimizing Sequence Queries in Database Systems. In *PODS'01*, 2001.

[13] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and Optimizing Sequence Queries in Database Systems. In *ACM TODS*, June 2004.

[14] F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB Conference*, Trondheim, Norway, January 2005.

[15] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD Conference*, Chicago, IL, June 2006.