

# CS 5300 Project #1

Jared Rainwater & Samuel K. Grush

October 31, 2017

## Contents

<b>1</b>	<b>The Compiler</b>	<b>1</b>
1.1	Grammar Rules . . . . .	2
1.2	Interpretation . . . . .	2
<b>2</b>	<b>Source Code</b>	<b>3</b>
2.1	src/ . . . . .	3
2.1.1	src/index.ts . . . . .	3
2.1.2	src/Main.tsx . . . . .	3
2.2	src/components . . . . .	5
2.2.1	src/components/QueryInput.tsx . . . . .	5
2.2.2	src/components/RelationsInput.tsx . . . . .	6
2.2.3	src/components/TestCase.tsx . . . . .	8
2.2.4	src/components/Tests.tsx . . . . .	11
2.2.5	src/components/tree.tsx . . . . .	13
2.3	src/parser . . . . .	14
2.3.1	src/parser/parsing.ts . . . . .	14
2.3.2	src/parser/relationalText.tsx . . . . .	14
2.3.3	src/parser/sqlToRel.ts . . . . .	19
2.3.4	src/parser/tests.ts . . . . .	29
2.3.5	src/parser/types.ts . . . . .	32
2.4	src/parser/peg . . . . .	39
2.4.1	src/parser/peg/sql.pegjs . . . . .	39
2.4.2	src/parser/peg/relations.pegjs . . . . .	47
2.5	src/query_tree . . . . .	48
2.5.1	src/query_tree/node.ts . . . . .	48
2.5.2	src/query_tree/operation.ts . . . . .	49
2.5.3	src/query_tree/parse.ts . . . . .	50

## 1 The Compiler

In order to parse SQL commands, we are using a parsing library called **PEG.js**, which allows us to express a/n SQL syntax as a *Parsing Expression Grammar* (PEG), and build that grammar into a JavaScript parser. The grammar was initially structured after Phoenix's SQL grammar, but generally follows PostgreSQL's syntax and the corresponding ANSI SQL standard.

## 1.1 Grammar Rules

*The grammar is defined in `src/parser/peg/sql.pegjs`.*

Parsing starts out with the `Statements` rule, which is a semicolon delimited list of SQL `Statements`. A `Statement` can be either a `Select` or `SelectPair`. `Select` is broken up into 6 clauses: `TargetClause`, `FromClause`, `WhereClause`, `GroupByClause`, `HavingClause` and `OrderByClause`. These correspond to all the possibilities of a valid SQL `Select` statement. A `SelectPair` is two separate `Select` clauses paired together with a “UNION”, “INTERSECT”, or “EXCEPT” set operation. You can also apply the “ALL” or “DISTINCT” modifier to the pair.

The `TargetClause` can have the optional “DISTINCT” or “ALL” modifier followed by “\*” (to allow everything) or a `TargetList`, a comma-delimited list of `TargetItems`. A `TargetItem` is a column-like specifier; it can be a relation name with “.” or an `Operand` with optional alias.

`FromClause` aliases `RelationList`, a list of comma-delimited relation-like fields, each of which may be a table name (with optional alias) or a `Join`. A `JOIN` is a pair of relation-like fields joined by a join-type (“CROSS”, “INNER”, “LEFT”, etc) followed by an optional join-condition (“ON Condition” or “USING (TargetList)”).

`WhereClause` and `HavingClause` are `Conditions`. The types of `Conditions` are: “OR” and “AND” (which join two `Conditions`); comparison, “LIKE”, and “BETWEEN” (which join two `Operands`); and “IN” and “EXISTS” (which take `Select`-like arguments).

`GroupByClause` is simply a `TargetList` like the target clause. `OrderByClause` is a comma-delimited list of `Operands`, each optionally with an ordering-condition (“ASC”, “DESC” “USING ...”).

An `Operand` is a `Term` optionally joined to other `Operands` by value operations (e.g. arithmetic or concatenation). A `Term` is a `Literal`, aggregate function, or column reference. `Literals` include numeric literals, booleans literals, and string literals (single-quoted).

A `Name`, which might refer to an operand or relation, is denoted by a bare-identifier (`/[a-z_][a-z0-9_]*` and not a `ReservedWord`) or any string quoted with double-quotes (“...”) or backticks (``...``).

Both comment forms are supported: starting with `--` and consuming the rest of the line, and C-style starting with `/*` and ending at `*/`. Both are permitted anywhere whitespace is.

The `ReservedWord` rule contains 340 keywords that the ISO/ANSI SQL:2008 standard states are **never allowed as identifiers**. This set is almost certainly overkill, as most SQL implementations only reserve a *small* fraction of it. It is also excessively large, making up over  $\frac{1}{3}$  of the grammar’s sourcecode and **90%** of the uncompressed compiled grammar.

## 1.2 Interpretation

*Classes and data structures discussed in this section defined in `src/parser/types.ts`.*

While parsing the grammar, the PEG.js parser calls JavaScript classes that correspond to SQL concepts. These classes include `SqlSelect`, `SqlJoin`, `SqlConditional`, `SqlLiteral`, etc. This generates an object-oriented data structure—resembling a tree—that represents the “SQL Structure”.

Once the SQL Structure is generated it can be converted into JavaScript classes that correspond to Relational Algebra concepts. These classes include `RelRestriction`, `RelProjection`, `RelJoin`, `RelConditional`, etc. This generates a data structure—more closely resembling a tree than before—that represents the “Relational Algebra Structure”.

*Top-level functions for parsing/conversion defined in `src/parser/parsing.ts`, with conversion implementation functions defined in `src/parser/sqlToRel.ts`.*

## 2 Source Code

All of this code is available at <https://github.com/SKGrush/sqlparse5300>

### 2.1 src/

#### 2.1.1 src/index.ts

```
1 import * as React from "react"
2 import * as ReactDOM from "react-dom"
3
4 import './styles/tests.scss'
5
6 import Main from './Main'
7
8 ReactDOM.render(
9   React.createElement(Main),
10  document.getElementById("content")
11 )
```

#### 2.1.2 src/Main.tsx

```
1 import * as React from "react"
2
3 import * as JSONPretty from 'react-json-pretty'
4
5 const Tracer = require('pegjs-backtrace')
6
7 import {Catalog} from 'parser/types'
8
9 import RelationsInput, {RelationsInputOutput} from './components/RelationsInput'
10
11 import QueryInput from './components/QueryInput'
12 import Tests from './components/Tests'
13 import TestCase from './components/TestCase'
14
15 export interface MainState {
16   queryInputText: string
17   status: string
18   queryJSON: any
19   relJSON: any
20   catalog: Catalog | null
21
22   debug: string
23 }
24
25 export default class Main extends React.Component<any, MainState> {
26   constructor(props: any) {
27     super(props)
28     this.state = {
29       queryInputText: "",
30       status: "",
31       catalog: null,
32       queryJSON: null,
33       relJSON: null,
34       debug: ""
```

```

35     }
36
37     this.onRelationsInputUpdate = this.onRelationsInputUpdate.bind(this)
38     this.onQueryInputUpdate = this.onQueryInputUpdate.bind(this)
39
40 }
41
42 onRelationsInputUpdate(output: RelationsInputOutput) {
43     if (output.error) {
44         this.setState({
45             catalog: null,
46             status: `Error Parsing Relations: ${output.error}`,
47             debug: output.traceback
48         })
49     } else {
50         this.setState({
51             catalog: output.catalog,
52             status: "Successfully Parsed Relations",
53             debug: ""
54         })
55     }
56 }
57
58 onQueryInputUpdate(text: string): void {
59     this.setState({
60         status: "Parsing Query...",
61         queryInputText: text,
62         queryJSON: null,
63         relJSON: null,
64         debug: ""
65     })
66
67 }
68
69 /*parseQuery(): void {
70     const {queryInputText, catalog} = this.state
71
72     let queryJSON
73     let relJSON
74
75     const tracer = new Tracer(queryInputText, {
76         useColor: false,
77         showTrace: true
78     })
79     try {
80         queryJSON = parseSql(queryInputText, {tracer})
81         let root = parseSQLToTree(queryJSON)
82         this.setState({
83             queryJSON,
84             root,
85             status: "Query parsed; Generating relational algebra..."
86         })
87     } catch (ex) {
88         const err: SqlSyntaxError = ex
89         this.setState({
90             status: `Error Parsing Query: ${err.message}`,
91             debug: tracer.getParseTreeString()
92         })
93         throw err

```

```

94     }
95
96     try {
97         relJSON = sqlToRelationalAlgebra(queryJSON, catalog as Catalog)
98         this.setState({
99             relJSON,
100             status: "Generated relational algebra"
101         })
102     } catch (ex) {
103         this.setState({
104             status: `Error Generating Relational Algebra:  ${ex.message}`,
105         })
106         throw ex
107     }
108 }*/
109
110 render() {
111     return (
112         <main id="main">
113             <RelationsInput onUpdate={this.onRelationsInputUpdate} />
114             <QueryInput
115                 onUpdate={this.onQueryInputUpdate}
116                 disabled={!this.state.catalog}
117             />
118             <div id="parse-status">{this.state.status}</div>
119             <div id="main-output">
120                 <TestCase
121                     catalog={this.state.catalog}
122                     queryInputText={this.state.queryInputText}
123                     doRun={true} // bad idea??
124                     anchor="main-test"
125                     name="Main Test"
126                 />
127                 <div id="debug-output" data-empty={!this.state.debug}>
128                     <pre><code>{this.state.debug}</code></pre>
129                 </div>
130             </div>
131             <hr />
132             <hr />
133             <Tests catalog={this.state.catalog} />
134         </main>
135     )
136 }
137 }

```

## 2.2 src/components

### 2.2.1 src/components/QueryInput.tsx

```

1  import * as React from "react"
2
3  export interface QueryInputProps {
4      onUpdate: (text: string) => void
5      disabled: boolean
6  }
7
8  export default class QueryInput extends React.Component<QueryInputProps, any> {

```

```

9   |   textInput: HTMLTextAreaElement
10  |
11  |   constructor(props: QueryInputProps) {
12  |       super(props)
13  |
14  |       this.onSubmit = this.onSubmit.bind(this)
15  |   }
16  |
17  |   onSubmit(e?) {
18  |       if (e) e.preventDefault()
19  |       console.info("Submitting:", this.textInput.value)
20  |       this.props.onUpdate(this.textInput.value)
21  |   }
22  |   render() {
23  |       return (
24  |           <div id="query-input-wrapper">
25  |               <textarea
26  |                   id="query-input"
27  |                   placeholder="Query..."
28  |                   cols={80}
29  |                   rows={10}
30  |                   ref={(input: HTMLTextAreaElement) => {this.textInput = input}}
31  |               />
32  |               <button
33  |                   disabled={this.props.disabled}
34  |                   onClick={this.onSubmit}
35  |               >Parse Query</button>
36  |           </div>
37  |       )
38  |   }
39  | }

```

## 2.2.2 src/components/RelationsInput.tsx

```

1   | import * as React from "react"
2   |
3   | const Tracer = require('pegjs-backtrace')
4   |
5   | import {parseRelations} from '../parser/parsing'
6   | import {Catalog} from '../parser/types'
7   |
8   | const DEFAULT_INPUT = `
9   | Sailors(sid:integer, sname:string, rating:integer, age:real)
10  | Boats(bid:integer, bname:string, color:string)
11  | Reserves(sid:integer, bid:integer, day:date)
12  | `
13  |
14  | export interface RelationsInputOutput {
15  |     catalog: Catalog | null
16  |     error: null | Error
17  |     traceback: '' | string
18  | }
19  |
20  | export interface RelationsInputProps {
21  |     onUpdate: (output: RelationsInputOutput) => void
22  | }
23  |

```

```

24 interface RelationsInputState {
25   catalog: Catalog | null
26   text: string
27 }
28
29 export default class RelationsInput extends React.Component<RelationsInputProps
    , RelationsInputState> {
30
31   constructor(props) {
32     super(props)
33     this.state = {
34       catalog: null,
35       text: DEFAULT_INPUT
36     }
37
38     this.run = this.run.bind(this)
39     this.onChange = this.onChange.bind(this)
40   }
41
42   run(e?) {
43     const text = this.state.text
44     if (e) e.preventDefault()
45
46     const tracer = new Tracer(text, {
47       useColor: false,
48       showTrace: true
49     })
50
51     let catalog: Catalog|null = null
52     try {
53       catalog = parseRelations(text, {tracer})
54       this.props.onUpdate({ catalog, error: null, traceback: '' })
55     } catch (ex) {
56       this.props.onUpdate({
57         catalog,
58         error: ex,
59         traceback: tracer.getParseTreeString()
60       })
61     }
62     this.setState({catalog})
63   }
64
65   onChange(event) {
66     this.setState({text: event.target.value})
67   }
68
69   render() {
70     return (
71       <div id="relations-input-wrapper">
72         <textarea
73           id="relations-input"
74           value={this.state.text}
75           cols={80}
76           rows={10}
77           onChange={this.onChange}
78         />
79         <button onClick={this.run}>Parse Relations</button>
80       </div>
81     )

```

```

82 |   }
83 | }

```

### 2.2.3 src/components/TestCase.tsx

```

1 | import * as React from "react"
2 | import * as JSONPretty from 'react-json-pretty'
3 | const Tracer = require('pegjs-backtrace')
4 |
5 | import {Catalog} from '../parser/types'
6 | import {parseSql, SqlSyntaxError, sqlToRelationalAlgebra} from '../parser/
   | parsing'
7 | import {htmlHLR} from '../parser/relationalText'
8 |
9 | import {Projection} from '../query_tree/operation'
10 | import Node from '../query_tree/node'
11 | import parseSQLToTree from '../query_tree/parse'
12 | import Tree from '../components/tree'
13 |
14 | interface TestCaseProps {
15 |   catalog: Catalog | null
16 |   queryInputText: string
17 |   doRun: boolean
18 |   anchor: string
19 |   name?: string
20 | }
21 |
22 | interface TestCaseState {
23 |   status: string
24 |   treeStatus: string
25 |   queryJSON: any
26 |   relAlJSON: any
27 |   root: Node | null
28 |   relAlHTML: JSX.Element | null
29 |   color: string
30 |   tscolor: string
31 |   debug: any
32 | }
33 |
34 | export default class TestCase extends React.Component<TestCaseProps,
   | TestCaseState> {
35 |   constructor(props) {
36 |     super(props)
37 |     this.state = this.initialState()
38 |     this.run = this.run.bind(this)
39 |   }
40 |
41 |   componentDidMount() {
42 |     this.propsReceived(this.props)
43 |   }
44 |
45 |   componentWillReceiveProps(newProps: TestCaseProps) {
46 |     this.propsReceived(newProps)
47 |   }
48 |
49 |   propsReceived(newProps: TestCaseProps) {
50 |     const {catalog, queryInputText, doRun} = this.props

```



```

51     if (newProps.catalog !== catalog ||
52         newProps.queryInputText !== queryInputText ||
53         newProps.doRun !== doRun
54     ) {
55         this.setState(this.initialState(), () => {
56             if (newProps.catalog && newProps.queryInputText && newProps.doRun)
57                 this.run(newProps)
58         })
59     }
60 }
61
62 initialState(): TestCaseState {
63     return {
64         status: 'init',
65         treeStatus: '',
66         queryJSON: null,
67         relAlJSON: null,
68         relAlHTML: null,
69         root: null,
70         color: 'currentcolor',
71         tscolor: 'currentcolor',
72         debug: ''
73     }
74 }
75
76 run(props: TestCaseProps = this.props) {
77
78     const catalog = props.catalog as Catalog
79
80     const tracer = new Tracer(props.queryInputText, {
81         useColor: false,
82         showTrace: true
83     })
84
85     let status = ''
86     let treeStatus = ''
87     let queryJSON = null
88     let relAlJSON = null
89     let relAlHTML = null
90     let root: Node | null = null
91     let color = 'currentcolor'
92     let tscolor = 'currentcolor'
93     let debug = ''
94
95     try {
96         queryJSON = parseSql(props.queryInputText, {tracer})
97         status = "SQL Scanned and Tokenized"
98         color = "green"
99     } catch (ex) {
100         if (ex instanceof SqlSyntaxError)
101             status = `Parser Syntax Error: ${ex.message}`
102         else
103             status = `Other Parser ${ex}`
104         console.error(ex)
105         color = "red"
106         debug = tracer.getParseTreeString()
107     }
108
109     if (queryJSON) {

```

```

110     try {
111         relAlJSON = sqlToRelationalAlgebra(queryJSON, catalog) as any
112         status = "SQL Parsed and converted to Relational Algebra"
113         color = "green"
114     } catch (ex) {
115         status = `Relational Algebra ${ex}`
116         color = "red"
117         console.error(ex)
118     }
119 }
120 if (relAlJSON) {
121     try {
122         relAlHTML = htmlHLR(relAlJSON)
123         status = "Relational Algebra rendered to HTML"
124         color = "green"
125     } catch (ex) {
126         status = `HTML Conversion Error: ${ex}`
127         color = "red"
128         console.error(ex)
129     }
130     try {
131         root = parseSQLToTree(/*relAlJSON*/queryJSON)
132         treeStatus = "Tree Generated"
133         tscolor = "green"
134     } catch (ex) {
135         treeStatus = `Tree Error: ${ex}`
136         tscolor = "red"
137         console.error(ex)
138     }
139 }
140
141 this.setState({
142     status,
143     treeStatus,
144     queryJSON,
145     relAlJSON,
146     relAlHTML,
147     root,
148     color,
149     tscolor,
150     debug
151 })
152 }
153
154 render() {
155     return (
156         <section id={this.props.anchor} className="testcase">
157             <hr />
158             <h3>{this.props.name} || {this.props.anchor}</h3>
159             <pre><code>{this.props.queryInputText}</code></pre>
160             <div className="testcase-status">
161                 <span style={{color: this.state.color}}>
162                     Status: {this.state.status} || "OK"
163                 </span>
164                 { this.state.treeStatus && (
165                     <span style={{color: this.state.tscolor}}>
166                         Tree Status: {this.state.treeStatus}
167                     </span>
168                 )}

```

```

169     </div>
170     <div className="testcase-inner">
171       <div className="relal-html" data-empty={!this.state.relAlHTML}>
172         <h4>Relational Algebra</h4>
173         {this.state.relAlHTML}
174       </div>
175       <div className="sql-json" data-empty={!this.state.queryJSON}>
176         <h4>SQL Structure</h4>
177         <JSONPretty json={this.state.queryJSON} />
178       </div>
179       <div className="relal-json" data-empty={!this.state.relAlJSON}>
180         <h4>Relational Algebra Structure</h4>
181         <JSONPretty json={this.state.relAlJSON} />
182       </div>
183       <div className="tree" data-empty={!this.state.root}>
184         <h4>Tree</h4>
185         { this.state.root &&
186           <Tree root={this.state.root} margin={10} />
187         }
188       </div>
189       <div className="traceback" data-empty={!this.state.debug}>
190         <h4>Error Traceback</h4>
191         <pre><code>{this.state.debug}</code></pre>
192       </div>
193     </div>
194   </section>
195 )
196 }
197 }

```

#### 2.2.4 src/components/Tests.tsx

```

1  import * as React from "react"
2
3  import {Catalog} from '../parser/types'
4  import TestCase from '../TestCase'
5  import {selectTests} from "../parser/tests"
6
7  export function getTestName(testStr: string) {
8    if (testStr.startsWith('-- '))
9      return testStr.split("\n", 1)[0].slice(2).trim()
10   return ''
11 }
12
13 interface TestsProps {
14   catalog: Catalog | null
15 }
16
17 interface TestsState {
18   catalog: Catalog | null
19   doRun: boolean
20   queryNames: string[]
21 }
22
23 export default class Tests extends React.Component<TestsProps, TestsState> {
24   constructor(props) {
25     super(props)

```

```

26   this.state = {
27     catalog: props.Catalog,
28     doRun: false,
29     queryNames: selectTests.map(getTestName)
30   }
31
32   this.run = this.run.bind(this)
33 }
34
35 componentWillReceiveProps(nextProps: TestsProps) {
36   const catalog = nextProps.catalog
37   if (catalog !== this.props.catalog)
38     this.setState({
39       catalog,
40       doRun: false
41     })
42 }
43
44 run(e?) {
45   if (e) e.preventDefault()
46   if (this.state.catalog)
47     this.setState({
48       doRun: true
49     })
50 }
51
52 render() {
53   return (
54     <div id="tests-div">
55       <h2>Test Cases</h2>
56       <button
57         onClick={this.run}
58         disabled={!this.state.catalog}
59       >Run Tests</button>
60       <nav id="tests-nav">
61         <ol>
62           {
63             this.state.queryNames.map((qName, idx) => {
64               const anchor = `#q${idx}`
65               return (
66                 <li key={anchor}>
67                   <a href={anchor}>{qName || anchor}</a>
68                 </li>
69               )
70             })
71           }
72         </ol>
73       </nav>
74       <div id="tests-list">
75         {
76           selectTests.map((testStr, idx) => (
77             <TestCase
78               queryInputText={testStr}
79               catalog={this.state.catalog}
80               doRun={this.state.doRun}
81               key={idx}
82               anchor={`#q${idx}`}
83               name={this.state.queryNames[idx] || undefined}
84             />

```

```

85         ))
86     }
87     </div>
88 </div>
89 )
90 }
91 }

```

### 2.2.5 src/components/tree.tsx

```

1  import * as React from 'react'
2  import Node from 'query_tree/node'
3  import '../styles/tree.scss'
4
5  interface TreeProps {
6      root: Node
7      margin: number
8  }
9
10 interface TreeState {
11 }
12
13 export default
14 class Tree extends React.Component<TreeProps, TreeState> {
15     render() {
16         const rows: JSX.Element[] = []
17         let node = this.props.root
18         let depth = 0
19         while (node) {
20             const row = <TreeRow
21                 node={node}
22                 key={depth}
23                 offset={this.props.margin * depth}/>
24             rows.push(row)
25             depth++
26             node = node.children[0]
27         }
28         return (
29             <div>
30                 {rows}
31             </div>
32         )
33     }
34 }
35
36 interface TreeRowProps {
37     offset: number
38     node: Node
39 }
40
41 interface TreeRowState {
42 }
43
44 class TreeRow extends React.Component<TreeRowProps, TreeRowState> {
45     render() {
46         return (
47             <div className="tree-row" style={{paddingLeft: this.props.offset}}>

```

```

48     {this.props.node.operation.name} ({this.props.node.operation.arguments.
      map(arg => {
49         return arg + ", "
50     })})
51   </div>
52 )
53 }
54 }

```

## 2.3 src/parser

### 2.3.1 src/parser/parsing.ts

```

1
2 import { parse as RelationParse } from './peg/relations'
3 import { parse as SqlParse } from './peg/sql'
4 export { SyntaxError as SqlSyntaxError } from './peg/sql'
5 import * as types from './types'
6 import {fromSqlSelect, fromSelectPair} from './sqlToRel'
7
8 export function parseRelations(input: string, args?): types.Catalog {
9     return types.Catalog.fromParse(RelationParse(input, args))
10 }
11
12 export function parseSql(input: string, args?) {
13     return SqlParse(input, args)
14 }
15
16 export function sqlToRelationalAlgebra(sqlStatements, catalog: types.Catalog) {
17     if (!Array.isArray(sqlStatements))
18         throw new Error("Expected SQL statements")
19     if (sqlStatements.length > 1)
20         throw new Error("Multiple statements not supported")
21
22     const TLStatement = sqlStatements[0]
23     if (TLStatement instanceof types.SqlSelect)
24         return fromSqlSelect(TLStatement, catalog)
25     else if (TLStatement instanceof types.SqlSelectPair)
26         return fromSelectPair(TLStatement, catalog)
27     else
28         throw new Error(`Unknown sqlToRelationalAlgebra arg ${TLStatement}`)
29 }

```

### 2.3.2 src/parser/relationalText.tsx

```

1 import * as React from 'react'
2
3 import * as types from './types'
4
5 export function getSymbol(input: string) {
6     switch (input) {
7         // passthroughs
8         case '||':
9         case '+':
10        case '-':

```

```

11     case '*':
12     case '/':
13     case '<':
14     case '>':
15         return input
16
17     case 'restriction':
18         return "ĩĈ"
19     case 'projection':
20         return "ĩă"
21     case 'rename':
22         return "ĩĀ"
23     case 'rename-divider':
24         return "ăĬ"
25
26     case 'union':
27         return "ăĬ"
28     case 'intersect':
29         return "ăĬ"
30     case 'except':
31         return "ăĬ"
32
33     case 'join':
34         return "ăĬ"
35     case 'left':
36     case 'ljoin':
37         return "ăĬ"
38     case 'right':
39     case 'rjoin':
40         return "ăĬ"
41     case 'cross':
42     case 'crossjoin':
43         return "ăĬ"
44     case 'divide':
45         return "ăĬ"
46
47     case 'eq':
48         return "="
49     case 'neq':
50         return "ăĬă"
51     case 'leq':
52         return "ăĬă"
53     case 'geq':
54         return "ăĬă"
55     case 'and':
56         return "ăĬă"
57     case 'or':
58         return "ăĬă"
59     case 'in':
60         return "ăĬă"
61     default:
62         throw new Error(`Unknown symbol name "${input}"`)
63     }
64 }
65
66 export function htmlRelRestriction(res: types.RelRestriction) {
67     const SYM = getSymbol('restriction')
68     const COND = htmlRelConditional(res.conditions)
69     const ARGS = htmlHLR(res.args)

```

```

70   return (
71     <span className="RelRestriction">
72       <span className="operator">{SYM}</span>
73       <sub className="condition">
74         {COND}
75       </sub>
76     (
77       <span className="HLR">
78         {ARGS}
79       </span>
80     )
81   </span>
82 )
83 }
84
85 export function htmlRelProjection(res: types.RelProjection) {
86   const SYM = getSymbol('projection')
87   const COLUMNS: Array<string|HTMLSpanElement> = []
88   res.columns.forEach((col, idx) => {
89     if (idx > 0)
90       COLUMNS.push(",")
91     if (col instanceof types.RelColumn)
92       COLUMNS.push(htmlRelColumn(col, idx))
93     else if ((col as any) instanceof types.RelFunction)
94       COLUMNS.push(htmlRelFunction(col, idx))
95     else
96       COLUMNS.push(col)
97   })
98   const ARGS = htmlHLR(res.args)
99   return (
100     <span className="RelProjection">
101       <span className="operator">{SYM}</span>
102       <sub className="columns">
103         {COLUMNS}
104       </sub>
105     (
106       <span className="HLR">
107         {ARGS}
108       </span>
109     )
110   </span>
111 )
112 }
113
114 export function htmlRelColumn(col: types.RelColumn, iter?: number) {
115   if (col.as) {
116     return (
117       <span className="RelColumn" key={iter}>
118         <span className="column-as">{col.as}</span>
119       </span>
120     )
121   }
122 }
123
124 if (!col.relation) {
125   return (
126     <span className="RelColumn" key={iter}>
127       <span className="column-name">{getName(col.target)}</span>
128     </span>

```



```

129   )
130   }
131
132   return (
133     <span className="RelColumn" key={iter}>
134       <span className="relation-name">{getName(col.relation)}</span>
135       .
136       <span className="column-name">{getName(col.target)}</span>
137     </span>
138   )
139 }
140
141 export function htmlRelFunction(func: types.RelFunction, idx?) {
142   const NAME = func.fname.toUpperCase()
143   const EXPR = func.expr === '*'
144     ? '*'
145     : htmlRelColumn(func.expr)
146
147   return (
148     <span className="RelFunction" key={idx}>
149       <span className="function-name">{NAME}</span>
150       (
151         {EXPR}
152       )
153     </span>
154   )
155 }
156
157 export function getName(thing) {
158   if (typeof(thing) === 'string')
159     return thing
160   if (thing instanceof types.RelRelation)
161     return thing.name
162   if (thing instanceof types.RelColumn)
163     return thing.as || htmlRelColumn(thing)
164   if (thing instanceof types.RelFunction)
165     return htmlRelFunction(thing as types.RelFunction)
166   if (thing instanceof types.Column)
167     return thing.name
168   console.info("getName", thing)
169   throw new Error("unexpected thing to getName")
170 }
171
172 export function htmlRelRename(ren: types.RelRename) {
173   const SYM = getSymbol('rename')
174   const INPUT = getName(ren.input)
175   const OUTPUT = ren.output
176   const ARGS = htmlHLR(ren.args as types.HighLevelRelationish)
177
178   return (
179     <span className="RelRename">
180       <span className="operator">{SYM}</span>
181       <sub className="condition">
182         {OUTPUT} {getSymbol('rename-divider')} {INPUT}
183       </sub>
184       (
185         <span className="HLR">
186           {ARGS}
187         </span>

```

```

188     )
189     </span>
190   )
191 }
192
193 export function htmlRelRelation(rel: types.RelRelation) {
194   const NAME = rel.name
195   return (
196     <span className="RelRelation">
197       {NAME}
198     </span>
199   )
200 }
201
202 export function htmlRelJoin(join: types.RelJoin) {
203   let joinSymbol
204   let cond
205   if (typeof(join.condition) === 'string') {
206     joinSymbol = getSymbol(join.condition)
207     cond = null
208   } else if (join.condition instanceof types.RelConditional) {
209     joinSymbol = getSymbol('join')
210     cond = htmlRelConditional(join.condition)
211   } else {
212     throw new Error(`unknown RelJoin condition ${join.condition}`)
213   }
214   const LHS = htmlHLR(join.lhs)
215   const RHS = htmlHLR(join.rhs)
216
217   return (
218     <span className="RelJoin">
219       {LHS}
220       <span className="operator">{joinSymbol}</span>
221       {
222         cond && (
223           <sub className="condition">
224             {cond}
225           </sub>
226         )
227       }
228       {RHS}
229     </span>
230   )
231 }
232
233 export function htmlRelOperation(op: types.RelOperation) {
234   const OPSYM = getSymbol(op.op)
235   const LHS = htmlRelOperand(op.lhs as any)
236   const RHS = htmlRelOperand(op.rhs as any)
237
238   return (
239     <span className="RelOperation">
240       {LHS}
241       <span className="operator">{OPSYM}</span>
242       {RHS}
243     </span>
244   )
245 }
246

```

```

247 export function htmlRelOperand(operand: types.RelOperandType) {
248   if (typeof(operand) === 'string')
249     return operand
250   if (operand instanceof types.RelFunction)
251     return htmlRelFunction(operand)
252   if (operand instanceof types.RelOperation)
253     return htmlRelOperation(operand)
254   if (operand instanceof types.RelColumn)
255     return htmlRelColumn(operand)
256   // throw new Error("Unexpected operand type")
257   return htmlHLR(operand)
258 }
259
260 export function htmlRelConditional(cond: types.RelConditional) {
261   const OPSYM = getSymbol(cond.operation)
262   const LHS = cond.lhs instanceof types.RelConditional
263     ? htmlRelConditional(cond.lhs)
264     : htmlRelOperand(cond.lhs)
265   const RHS = cond.rhs instanceof types.RelConditional
266     ? htmlRelConditional(cond.rhs)
267     : ( cond.rhs instanceof Array
268       ? cond.rhs.map(htmlRelOperand)
269       : htmlRelOperand(cond.rhs)
270     )
271
272   return (
273     <span className="RelConditional">
274       <span className="lhs">
275         {LHS}
276       </span>
277       <span className="operator">{OPSYM}</span>
278       <span className="rhs">
279         {RHS}
280       </span>
281     </span>
282   )
283 }
284
285 export function htmlHLR(hlr: types.HighLevelRelationish) {
286   if (hlr instanceof types.RelRestriction)
287     return htmlRelRestriction(hlr)
288   if (hlr instanceof types.RelProjection)
289     return htmlRelProjection(hlr)
290   if (hlr instanceof types.RelRename)
291     return htmlRelRename(hlr)
292   if (hlr instanceof types.RelOperation)
293     return htmlRelOperation(hlr)
294   if (hlr instanceof types.RelRelation)
295     return htmlRelRelation(hlr)
296   if (hlr instanceof types.RelJoin)
297     return htmlRelJoin(hlr)
298   console.error("unknown HLR:", hlr)
299   throw new Error("Unknown type passed to htmlHLR")
300 }

```

### 2.3.3 src/parser/sqlToRel.ts

```

1
2 import * as types from './types'
3
4 type ColumnValueType = types.RelColumn | types.RelFunction | string
5
6 type RelationLookup = Map<string, types.RelRelation>
7
8 /* bubble a join/relation up to the calling function, also returning
9    the 'realOperation' that took place */
10 class BubbleUp<T> {
11     realOperation: T
12     relationish: types.HighLevelRelationish
13
14     constructor(realOp: T, relationish: types.HighLevelRelationish) {
15         this.realOperation = realOp
16         this.relationish = relationish
17     }
18 }
19
20 class RenameBubbleUp {
21     target: ColumnValueType
22     output: string
23
24     constructor(target: ColumnValueType, output: string) {
25         this.target = target
26         this.output = output
27     }
28 }
29
30 class ColumnLookup {
31     readonly map: Map<string, types.RelColumn[]>
32     readonly catalog: types.Catalog
33     readonly relations: RelationLookup
34
35     constructor(catalog: types.Catalog, relations: RelationLookup, init?) {
36         this.map = new Map(init)
37         this.catalog = catalog
38         this.relations = relations
39     }
40
41     addAlias(name: string, target: types.RelColumn) {
42         const cols = this.map.get(name)
43         if (!cols)
44             this.map.set(name, [target])
45         else
46             cols.push(target)
47     }
48
49     lookup(columnName: string, relationName?: string, as?: string): types.
        RelColumn {
50         if (relationName) {
51             // column references a relation
52             if (!this.relations.has(relationName)) {
53                 throw new Error(`Unknown relation "${relationName}"`)
54             }
55             const relation = this.relations.get(relationName) as types.RelRelation
56             const catRelation = this.catalog.relations.get(relation.name) as types.
                Relation
57             // if(!catRelation)

```

```

58 // throw new Error(`${relationName} not in catalog`)
59 if (catRelation.columns.has(columnName))
60     return new types.RelColumn(relation,
61                                 catRelation.columns.get(columnName) as types
62                                     .Column,
63                                     as)
64 else
65     throw new Error(`${catRelation.name} doesn't contain ${columnName}`)
66 } else {
67     // implicit relation reference
68     if (this.map.has(columnName)) {
69         // already in the map
70         const cols = this.map.get(columnName) as types.RelColumn[]
71         if (cols.length > 1)
72             throw new Error(`Ambiguous column name reference "${columnName}"`)
73         return cols[0].alias(as)
74     }
75     // not in map; search for columnName
76     console.group()
77     console.info(`Searching for ${columnName}`)
78     for (const val of this.relations.values()) {
79         // if (!this.catalog.relations.has(val.name)) {
80         //     throw new Error(`${val.name} not in catalog`)
81         // }
82         const catRel = this.catalog.relations.get(val.name) as types.Relation
83         console.info(`${val.name} in catalog, looking for ${columnName}`)
84         if (!catRel.columns.has(columnName))
85             continue
86         console.info(`found`)
87         console.groupEnd()
88         const col = catRel.columns.get(columnName) as types.Column
89         return new types.RelColumn(val, col, as)
90     }
91     console.info(`not found`)
92     console.groupEnd()
93     throw new Error(`Unknown column ${columnName}`)
94 }
95 }
96 }
97 }
98
99 function _joinArgHelper(hs: types.SqlJoin | types.SqlRelation,
100                        relations: RelationLookup,
101                        columns: ColumnLookup,
102                        catalog: types.Catalog,
103                        arg: types.SqlJoin,
104                        side): types.RelRelationish {
105     if (hs instanceof types.SqlJoin)
106         return fromJoin(hs, relations, columns, catalog)
107     else if (hs instanceof types.SqlRelation)
108         return fromRelation(hs, relations, columns, catalog) as types.RelRelation
109     console.error(`bad join arg ${side}`, arg, "lookup:", relations)
110     throw new Error("Bad join argument lhs")
111 }
112
113 function fromJoin(arg: types.SqlJoin,
114                  relations: RelationLookup,
115                  columns: ColumnLookup,

```

```

116         catalog: types.Catalog): types.RelJoin {
117     const lhs = _joinArgHelper(arg.lhs, relations, columns, catalog, arg, 'left')
118     const rhs = _joinArgHelper(arg.rhs, relations, columns, catalog, arg, 'right'
119     )
120     let cond: any = null
121     if (arg.condition) {
122         if (arg.condition instanceof types.SqlConditional)
123             cond = fromConditional(arg.condition, relations, columns, catalog)
124         else if (Array.isArray(arg.condition) && arg.condition.length === 2)
125             cond = fromTargetList(arg.condition[1], relations, columns, catalog)
126         else {
127             console.error("bad conditional", arg, "lookup:", relations)
128             throw new Error("bad conditional")
129         }
130     } else {
131         switch (arg.joinType) {
132             case "join":
133             case null:
134                 cond = "cross"
135                 break
136             case "leftouter":
137                 cond = "left"
138                 break
139             case "rightouter":
140                 cond = "right"
141                 break
142             case "fullouter":
143                 throw new Error("full outer join not supported")
144             // case "natural" | "equi" | null:
145         }
146     }
147     const J = new types.RelJoin(lhs, rhs, cond)
148     return J
149 }
150
151 function fromColumn(arg: types.SqlColumn,
152     relations: RelationLookup,
153     columns: ColumnLookup,
154     catalog: types.Catalog
155 ): RenameBubbleUp | ColumnValueType {
156     const alias = arg.alias
157     let target
158     if (arg.target instanceof types.SqlColumn) {
159         // column of column; either rename it or return target
160         target = fromColumn(arg.target, relations, columns, catalog)
161         if (!alias)
162             console.warn("Why double column?")
163         else if (target instanceof RenameBubbleUp) {
164             console.error("Double rename; arg,target =", arg, target)
165             throw new Error("Double rename not supported")
166         }
167     } else if (typeof(arg.target) === 'string') {
168         // column based on a name
169         target = columns.lookup(arg.target,
170             arg.relation || undefined,
171             arg.as || undefined)
172     } else if (arg.target instanceof types.SqlLiteral) {
173         target = fromLiteral(arg.target)

```

```

174 } else if (arg.target instanceof types.SqlAggFunction) {
175     target = fromAggFunction(arg.target, relations, columns, catalog)
176 } else {
177     throw new Error("Unexpected type in column")
178 }
179
180 if (alias) {
181     columns.addAlias(alias, target)
182     return new RenameBubbleUp(target, alias)
183 }
184 return target
185 }
186
187 function fromTargetList(targetColumns: types.SqlColumn[],
188                         relationLookup: RelationLookup,
189                         columnLookup: ColumnLookup,
190                         catalog: types.Catalog
191 ): [ColumnValueType[], RenameBubbleUp[]] {
192     console.info("fromTargetList:", targetColumns)
193     const renames: RenameBubbleUp[] = []
194     const cols = targetColumns.map((colarg) => {
195         const col = fromColumn(colarg,
196                                relationLookup,
197                                columnLookup,
198                                catalog)
199         if (col instanceof RenameBubbleUp) {
200             renames.push(col)
201             return col.target
202         }
203         return col
204     })
205     return [cols, renames]
206 }
207
208 function fromRelation(arg: types.SqlRelation,
209                      relations: RelationLookup,
210                      columns: ColumnLookup,
211                      catalog: types.Catalog): types.RelRename | types.
212                      RelRelation | types.RelJoin {
213     if (typeof(arg.target) === 'string') {
214         let relat
215         if (relations.has(arg.target))
216             relat = relations.get(arg.target)
217         else if (catalog.relations.has(arg.target)) {
218             relat = new types.RelRelation(arg.target)
219             relations.set(arg.target, relat)
220         } else {
221             console.error(`Unknown relation ${arg.target}`, arg, relations)
222             throw new Error(`Unknown relation ${arg.target}`)
223         }
224
225         if (arg.alias) {
226             const ren = new types.RelRename(relat, arg.alias, relat)
227             relations.set(arg.alias, relat)
228             return ren
229         }
230         return relat
231     } else if (arg.target instanceof types.SqlRelation) {
232         const relat = fromRelation(arg.target, relations, columns, catalog) as

```

```

    types.RelRelation
232     if (!arg.alias)
233         return relat
234     const ren = new types.RelRename(relat, arg.alias, relat)
235     relations.set(arg.alias, relat)
236     return ren
237 } else if (arg.target instanceof types.SqlJoin) {
238     const J = fromJoin(arg.target, relations, columns, catalog)
239     if (!arg.alias)
240         return J
241     else
242         throw new Error("Renaming joins not supported ")
243     // const ren = new types.RelRename()
244 } else {
245     console.error("bad arg.target type", arg, "lookup:", relations)
246     throw new Error("bad arg.target type")
247 }
248 }
249
250 function fromRelationList(arg: types.RelationList,
251                           relations: RelationLookup,
252                           columns: ColumnLookup,
253                           catalog: types.Catalog) {
254     if (arg instanceof types.SqlRelation)
255         return fromRelation(arg, relations, columns, catalog)
256     else
257         return fromJoin(arg, relations, columns, catalog)
258 }
259
260 function fromLiteral(lit: types.SqlLiteral) {
261     switch (lit.literalType) {
262         case 'string':
263             return `${lit.value}`
264         case 'number':
265         case 'boolean':
266         case 'null':
267             return String(lit.value)
268         default:
269             throw new Error(`Unknown literal type ${lit.literalType} for ${lit.value}`)
270     }
271 }
272
273 function fromAggFunction(agg: types.SqlAggFunction,
274                           rels: RelationLookup,
275                           cols: ColumnLookup,
276                           cata: types.Catalog) {
277     switch (agg.fname) {
278         case 'count':
279             if (agg.expr === '*' || (agg.expr as types.TargetClause).targetlist === '*')
280                 return new types.RelFunction('count', '*')
281             else
282                 throw new Error("Counting columns not supported")
283         case 'avg':
284         case 'max':
285         case 'min':
286         case 'sum':
287             if (!(agg.expr instanceof types.SqlColumn))

```



```

288     throw new Error(`non-column arguments to aggregates not supported`)
289     const expr = fromColumn(agg.expr, rels, cols, cata) as types.RelColumn
290     return new types.RelFunction(agg.fname, expr)
291   default:
292     throw new Error(`Unknown aggregate function ${agg.fname}`)
293   }
294 }
295
296 function fromOperation(arg: types.SqlOperation,
297                       rels: RelationLookup,
298                       cols: ColumnLookup,
299                       cata: types.Catalog) {
300   const lhs = _condArgHelper(arg.lhs, rels, cols, cata)
301   const rhs = _condArgHelper(arg.rhs, rels, cols, cata)
302   return new types.RelOperation(arg.op, lhs, rhs)
303 }
304
305 /* takes an Operand argument */
306 function _condArgHelper(hs, rels, cols, cata) {
307   if (hs instanceof Array)
308     return fromTargetList(hs, rels, cols, cata)[0]
309   if (hs instanceof types.SqlConditional)
310     return fromConditional(hs, rels, cols, cata)
311   else if (hs instanceof types.SqlSelect)
312     return fromSqlSelect(hs, cata)
313   // Operand
314   else if (hs instanceof types.SqlLiteral)
315     return fromLiteral(hs)
316   else if (hs instanceof types.SqlAggFunction)
317     return fromAggFunction(hs, rels, cols, cata)
318   else if (hs instanceof types.SqlColumn)
319     return fromColumn(hs, rels, cols, cata)
320   else if (hs instanceof types.SqlOperation)
321     return fromOperation(hs, rels, cols, cata)
322   else
323     throw new Error(`Unknown conditional arg type ${hs}`)
324 }
325
326 function _handleSubquery(arg, lhs, op, relations, columns, catalog) {
327
328   const tmpRhs = (arg.rhs instanceof types.SqlSelectPair)
329     ? fromSelectPair(arg.rhs, catalog)
330     : fromSqlSelect(arg.rhs, catalog)
331
332   if (op === 'in')
333     op = 'eq'
334
335   // lhs = check-against
336   // rhs = Selectish
337   if (!(tmpRhs instanceof types.RelProjection))
338     throw new Error("'in' subqueries must select columns")
339
340   const rhsTarget = tmpRhs.columns
341
342   let conditional: types.RelConditional
343   if (rhsTarget.length > 1)
344     conditional = rhsTarget.reduce((L, R) =>
345       new types.RelConditional(op, L, R), lhs)
346   else

```

```

347     conditional = new types.RelConditional(op, lhs, rhsTarget[0])
348
349     return new BubbleUp<types.RelConditional>(conditional, tmpRhs.args)
350 }
351
352 function fromConditional(arg: types.SqlConditional,
353                         relations: RelationLookup,
354                         columns: ColumnLookup,
355                         catalog: types.Catalog
356 ): types.RelConditional | BubbleUp<types.RelConditional> {
357     let binOp = true
358     let op: types.ThetaOp
359     switch (arg.operation) {
360         case 'not':
361         case 'isnull':
362         case 'exists':
363             binOp = false
364             // break
365         /* binary ops */
366         case 'like':
367         case 'between':
368             throw new Error(`${arg.operation} condition not yet supported`)
369
370         case 'or':
371         case 'and':
372         case 'in':
373         case '<':
374         case '>':
375             op = arg.operation
376             break
377         case '<>':
378         case '!=':
379             op = 'neq'
380             break
381         case '<=':
382             op = 'leq'
383             break
384         case '>=':
385             op = 'geq'
386             break
387         case '=':
388             op = 'eq'
389             break
390         default:
391             throw new Error(`Unknown op "${arg.operation}"`)
392     }
393     let lhs = _condArgHelper(arg.lhs, relations, columns, catalog)
394     if (lhs instanceof RenameBubbleUp) {
395         lhs = lhs.target
396     }
397
398     if (op === 'in' && arg.rhs instanceof Array) {
399         const rs = arg.rhs.map((R) => {
400             const tcond = _condArgHelper(R, relations, columns, catalog)
401             if (tcond instanceof RenameBubbleUp)
402                 return tcond.target
403             return tcond
404         })
405         const cond = new types.RelConditional('in', lhs, rs)

```

```

406     if (arg.not)
407         throw new Error("'not' conditional is not supported")
408     return cond
409 }
410 if (arg.rhs instanceof types.SqlSelect ||
411     arg.rhs instanceof types.SqlSelectPair) {
412     return _handleSubquery(arg, lhs, op, relations, columns, catalog)
413 }
414 if (op === 'in') {
415     throw new Error("'in' argument should be array or subquery")
416 }
417
418 if (!binOp)
419     throw new Error("unary operators not supported")
420 let rhs = _condArgHelper(arg.rhs, relations, columns, catalog)
421 if (rhs instanceof RenameBubbleUp)
422     rhs = rhs.target
423
424 const condit = new types.RelConditional(op, lhs, rhs)
425
426 if (arg.not)
427     throw new Error("'not' conditional is not supported")
428 return condit
429 }
430
431 function fromOrderings(orderings, rels, cols, cata) {
432     if (!orderings || !orderings.length)
433         return null
434     return orderings.map(([col, cond]) => {
435         const column = fromColumn(col, rels, cols, cata)
436         if (column instanceof RenameBubbleUp)
437             return [column.target, cond]
438         return [column, cond]
439     })
440 }
441
442 export function fromSelectPair(selPair: types.SqlSelectPair,
443                               catalog: types.Catalog) {
444     const lhs = fromSqlSelect(selPair.lhs, catalog)
445     let rhs
446     if (selPair.rhs instanceof types.SqlSelect)
447         rhs = fromSqlSelect(selPair.rhs, catalog)
448     else
449         rhs = fromSelectPair(selPair.rhs, catalog)
450
451     if (lhs instanceof types.RelProjection &&
452         rhs instanceof types.RelProjection) {
453         if (lhs.columns.length !== rhs.columns.length)
454             throw new Error(`Joining on unequal degrees: ` +
455                             `${lhs.columns.length} vs ${rhs.columns.length}`)
456         const newLhs = lhs.args
457         const newRhs = rhs.args
458         const newColumns = lhs.columns
459         const args = new types.RelOperation(selPair.pairing, newLhs, newRhs)
460         return new types.RelProjection(newColumns, args)
461     }
462
463     const operation = new types.RelOperation(selPair.pairing, lhs, rhs)
464     return operation

```

```

465 }
466
467 function _renameReducer(arg: types.HighLevelRelationish, ren: RenameBubbleUp) {
468   return new types.RelRename(ren.target, ren.output, arg)
469 }
470
471 function applyRenameBubbleUps(renames: RenameBubbleUp[],
472                               args: types.HighLevelRelationish) {
473   return renames.reduce(_renameReducer, args)
474 }
475
476 export function fromSqlSelect(select: types.SqlSelect, catalog: types.Catalog)
477 {
478   // map names to the actual instances
479   const relations = new Map()
480   const columns = new ColumnLookup(catalog, relations)
481
482   let fromClause: types.HighLevelRelationish
483     = fromRelationList(select.from, relations, columns, catalog)
484
485   let targetColumns
486   let renames: RenameBubbleUp[] = []
487   if (select.what.targetlist === '*')
488     targetColumns = '*'
489   else {
490     [targetColumns, renames] = fromTargetList(select.what.targetlist,
491                                              relations,
492                                              columns,
493                                              catalog)
494   }
495
496   // const whereClause = select.where
497   //   ? fromConditional(select.where, relations, columns, catalog)
498   //   : null
499   let whereClause: any = null
500   if (select.where) {
501     whereClause = fromConditional(select.where, relations, columns, catalog)
502     if (whereClause instanceof BubbleUp) {
503       fromClause = new types.RelJoin(fromClause, whereClause.relationish, '
504         cross')
505       whereClause = whereClause.realOperation as types.RelConditional
506     }
507   }
508
509   if (renames.length) {
510     fromClause = applyRenameBubbleUps(renames, fromClause)
511   }
512
513   const groupBy = select.groupBy
514     ? fromTargetList(select.groupBy, relations, columns, catalog)
515     : null
516
517   const having = select.having
518     ? fromConditional(select.having, relations, columns, catalog)
519     : null
520
521   const orderBy = fromOrderings(select.orderBy, relations, columns, catalog)

```

```

522     const Rest = whereClause
523         ? new types.RelRestriction(whereClause, fromClause)
524         : fromClause
525
526     const Proj = targetColumns === '*'
527         ? Rest
528         : new types.RelProjection(targetColumns, Rest)
529
530     return Proj
531 }

```

### 2.3.4 src/parser/tests.ts

```

1
2 export const selectTests = [
3
4   `-- Query 2a
5   SELECT      S.sname
6   FROM        Sailors AS S, Reserves AS R
7   WHERE       S.sid=R.sid AND R.bid=103`,
8
9   `-- Query 2b
10  SELECT      S.sname
11  FROM        Sailors AS S, Reserves AS R, Boats AS B
12  WHERE       S.sid=R.sid AND R.bid=B.bid AND B.color=âŽžredâŽž`,
13
14  `-- Query 2c
15  SELECT      sname
16  FROM        Sailors, Boats, Reserves
17  WHERE       Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
18  Boats.color=âŽžredâŽž
19  UNION
20  SELECT      sname
21  FROM        Sailors, Boats, Reserves
22  WHERE       Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
23  Boats.color='green'`,
24
25  `-- Query 2d (invalid)
26  -- unescaped reserve word 'day', invalid reference 'R.rating'
27  SELECT      S.sname
28  FROM        Sailors AS S, Reserves AS R
29  WHERE       R.sid = S.sid AND R.bid = 100 AND R.rating > 5 AND R.day =
30  âŽž8/9/09âŽž`,
31
32  `-- Modified Query2d (invalid)
33  -- still unknown reference 'R.rating'
34  SELECT      S.sname
35  FROM        Sailors AS S, Reserves AS R
36  WHERE       R.sid = S.sid AND R.bid = 100 AND R.rating > 5 AND R.`day` =
37  âŽž8/9/09âŽž`,
38
39  `-- Query 2e
40  SELECT      sname
41  FROM        Sailors, Boats, Reserves
42  WHERE       Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
43  Boats.color=âŽžredâŽž
44  INTERSECT

```

```

45 SELECT      sname
46 FROM        Sailors, Boats, Reserves
47 WHERE       Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
48 Boats.color=âĀĲgreenâĀĲ`,
49
50 `-- Query 2f (invalid)
51 -- illegal identifier '2color' of B
52 SELECT      S.sid
53 FROM        Sailors AS S, Reserves AS R, Boats AS B
54 WHERE       S.sid=R.sid AND R.bid=B.bid AND B.color=âĀĲredâĀĲ
55 EXCEPT
56 SELECT      S2.sid
57 FROM        Sailors AS S2, Reserves AS R2, Boats AS B2
58 WHERE       S2.sid=R2.sid AND R2.bid=B2.bid AND B.2color=âĀĲgreenâĀĲ`,
59
60 `-- Modified Query 2f
61 SELECT      S.sid
62 FROM        Sailors AS S, Reserves AS R, Boats AS B
63 WHERE       S.sid=R.sid AND R.bid=B.bid AND B.color=âĀĲredâĀĲ
64 EXCEPT
65 SELECT      S2.sid
66 FROM        Sailors AS S2, Reserves AS R2, Boats AS B2
67 WHERE       S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color=âĀĲgreenâĀĲ`,
68
69 `-- Query 2g (invalid)
70 -- unknown reference 'Reserve'
71 SELECT      S.sname
72 FROM        Sailors AS S
73 WHERE       S.sid IN ( SELECT      R.sid
74                        FROM        Reserve AS R
75                        WHERE       R.bid = 103)^`,
76
77 `-- Modified Query 2g
78 SELECT      S.sname
79 FROM        Sailors AS S
80 WHERE       S.sid IN ( SELECT      R.sid
81                        FROM        Reserves AS R
82                        WHERE       R.bid = 103)^`,
83
84 `-- Query 2h (invalid)
85 -- unknown reference 'Reserve'
86 SELECT      S.sname
87 FROM        Sailors AS S
88 WHERE       S.sid IN ((SELECT      R.sid
89                        FROM        Reserve AS R, Boats AS B
90                        WHERE       R.bid = B.bid AND B.color = âĀĲredâĀĲ)
91                     INTERSECT
92                     (SELECT      R2.sid
93                        FROM        Reserve AS R2, Boats AS B2
94                        WHERE       R2.bid = B2.bid AND B2.color = âĀĲgreenâĀĲ)))`,
95
96 `-- Modified Query 2h
97 SELECT      S.sname
98 FROM        Sailors AS S
99 WHERE       S.sid IN ((SELECT      R.sid
100                        FROM        Reserves AS R, Boats AS B
101                        WHERE       R.bid = B.bid AND B.color = âĀĲredâĀĲ)
102                     INTERSECT
103                     (SELECT      R2.sid

```

```

104         FROM      Reserves AS R2, Boats AS B2
105         WHERE      R2.bid = B2.bid AND B2.color = 'green'))`,
106
107 `-- Query 2i (invalid)
108 -- bad inner condition string, also unknown reference 'R'
109 SELECT      S.sname
110 FROM        Sailors AS S
111 WHERE       S.age > ( SELECT      MAX (S2.age)
112                        FROM        Sailors S2
113                        WHERE       R.sid = S2.rating = 10)` ,
114
115 `-- Modified Query 2i
116 SELECT      S.sname
117 FROM        Sailors AS S
118 WHERE       S.age > ( SELECT      MAX (S2.age)
119                        FROM        Sailors S2
120                        WHERE       S2.rating = 10)` ,
121
122 `-- Query 2j
123 SELECT      B.bid, Count (*) AS reservationcount
124 FROM        Boats B, Reserves R
125 WHERE       R.bid=B.bid AND B.color = 'red'
126 GROUP BY   B.bid`,
127
128 `-- Query 2k
129 SELECT      B.bid, Count (*) AS reservationcount
130 FROM        Boats B, Reserves R
131 WHERE       R.bid=B.bid AND B.color = 'red'
132 GROUP BY   B.bid
133 HAVING      B.color = 'red'`,
134
135 `-- Query 2l (invalid)
136 -- typo "SLECT", misuse of nonstandard 'contains' WHERE predicate, 'Sname'
137 SELECT      Sname
138 FROM        Sailors
139 WHERE       Sailor.sid IN (SELECT      Reserves.bid, Reserves.sid
140                                FROM      Reserves
141                                CONTAINS
142                                (SLECT Boats.bid
143                                FROM      Boats
144                                WHERE Boats.name = 'interlake' ) )`,
145
146 `-- Modified Query 2l (invalid, system-specific)
147 -- unknown reference 'Sname'
148 SELECT      Sname
149 FROM        Sailors
150 WHERE       Sailor.sid IN (SELECT      Reserves.bid, Reserves.sid
151                                FROM      Reserves
152                                WHERE     EXISTS (
153                                SELECT Boats.bid
154                                FROM      Boats
155                                WHERE Boats.name = 'interlake'
156                                AND Boats.bid = Reserves.bid ) )`,
157
158 `-- Query 2m (invalid)
159 -- Bad TargetList
160 SELECT      S.rating, Ave (S.age) As average
161 FROM        Sailors S
162 WHERE       S.age > 18

```

```

163 GROUP BY S.rating
164 HAVING Count (*) > 1`,
165
166 `-- Modified Query 2m
167 SELECT S.rating, Avg (S.age) As average
168 FROM Sailors S
169 WHERE S.age > 18
170 GROUP BY S.rating
171 HAVING Count (*) > 1`
172 ]

```

### 2.3.5 src/parser/types.ts

```

1
2 export const LITERAL_TYPE = "literal"
3 export const COLUMN_TYPE = "column"
4 export const JOIN_TYPE = "join"
5 export const RELATION_TYPE = "relation"
6 export const CONDITIONAL_TYPE = "conditional"
7 export const AGGFUNCTION_TYPE = "aggfunction"
8 export const OPERATION_TYPE = "operation"
9 export const SELECTCLAUSE_TYPE = "selectclause"
10 export const TARGETCLAUSE_TYPE = "targetclause"
11 export const SELECTPAIR_TYPE = "selectpair"
12
13 export const REL_RESTRICTION_TYPE = "restriction"
14 export const REL_PROJECTION_TYPE = "projection"
15 export const REL_RENAME_TYPE = "rename"
16
17 export const REL_RELATION_TYPE = "relrelation"
18 export const REL_COLUMN_TYPE = "relcolumn"
19 export const REL_CONDITIONAL_TYPE = "relconditional"
20 export const REL_JOIN_TYPE = "reljoin"
21 export const REL_FUNCTION_TYPE = "relfunt"
22 export const REL_OPERATION_TYPE = "relop"
23
24 /**
25  * IFF rhs is non-empty, run reduce using f on rhs initialized by lhs.
26  * Else return lhs
27  */
28 export function reduceIfRHS(lhs: any, rhs: any[], f: (L, R) => any) {
29   if (rhs.length)
30     return rhs.reduce(f, lhs)
31   return lhs
32 }
33
34 export class Catalog {
35
36   static fromParse(relations: Array<[string, Array<[string, string]>]> >) {
37     const rels = new Map()
38     relations.forEach((ele) => {
39       const [tname, cols] = ele
40       const columnMap = new Map()
41       cols.forEach((col) => {
42         columnMap.set(col[0], new Column(col[0], col[1]))
43       })
44       rels.set(tname, new Relation(tname, columnMap))

```



```

45     })
46     return new Catalog(rels)
47 }
48
49 relations: Map<string, Relation>
50
51 constructor(relations: Map<string, Relation>) {
52     this.relations = relations
53 }
54 }
55
56 export class Relation {
57     name: string
58     columns: Map<string, Column>
59
60     constructor(name: string, columns: Map<string, Column>) {
61         this.name = name
62         this.columns = columns
63     }
64 }
65
66 export class Column {
67     name: string
68     typ: string
69
70     constructor(name: string, typ: string) {
71         this.name = name
72         this.typ = typ
73     }
74 }
75
76 export type JOINSTRING = "join"           // "," | "JOIN" | "CROSS JOIN"
77                       | "equi"           // "INNER JOIN" | "JOIN ... USING"
78                       | "natural"       // "NATURAL JOIN"
79                       | "leftouter"     // "LEFT [OUTER] JOIN"
80                       | "rightouter"    // "RIGHT [OUTER] JOIN"
81                       | "fullouter"     // "FULL [OUTER] JOIN"
82
83 type OrderingCondition = "asc" | "desc" | "<" | ">"
84 type Ordering = [SqlColumn, OrderingCondition]
85
86 export type RelationList = SqlRelation | SqlJoin
87 type TargetList = SqlColumn[]
88
89 export interface TargetClause {
90     type: "targetclause"
91     spec: "distinct" | "all" | null
92     targetlist: "*" | TargetList
93 }
94
95 export class SqlLiteral {
96     readonly type = LITERAL_TYPE
97     literalType: 'string' | 'number' | 'boolean' | 'null'
98     value: string | number | boolean | null
99
100     constructor(literalType: 'string' | 'number' | 'boolean' | 'null',
101                 value: string | number | boolean | null) {
102         this.literalType = literalType
103         this.value = value

```

```

104   }
105 }
106
107 export type SqlSelectish = SqlSelect | SqlSelectPair
108 export type PairingString = 'union' | 'intersect' | 'except'
109 export type PairingCondition = 'all' | 'distinct' | null
110
111 export class SqlSelectPair {
112   readonly type = SELECTPAIR_TYPE
113   pairing: PairingString
114   condition: PairingCondition
115   lhs: SqlSelect
116   rhs: SqlSelectish
117
118   constructor(pairing: PairingString,
119               condition: PairingCondition,
120               lhs: SqlSelect,
121               rhs: SqlSelectish) {
122     this.pairing = pairing
123     this.condition = condition || null
124     this.lhs = lhs
125     this.rhs = rhs
126   }
127 }
128
129 export class SqlSelect {
130   readonly SELECTCLAUSE_TYPE
131   what: TargetClause
132   from: RelationList
133   where: SqlConditional | null
134   groupBy: TargetList | null
135   having: SqlConditional | null
136   orderBy: Ordering[] | null
137
138   constructor(what: TargetClause,
139               from: RelationList,
140               where: SqlConditional | null,
141               groupBy: TargetList | null,
142               having: SqlConditional | null,
143               orderBy: Ordering[] | null) {
144     this.what = what
145     this.from = from
146     this.where = where
147     this.groupBy = groupBy
148     this.having = having
149     this.orderBy = orderBy
150   }
151 }
152
153 export type SqlOperandType = SqlLiteral | SqlAggFunction | SqlColumn |
154                               SqlOperation | string
155
156 export class SqlColumn {
157   readonly type = COLUMN_TYPE
158   relation: string | null
159   target: SqlOperandType
160   as: string | null
161   alias: string | null
162 }

```

```

163     constructor(relation: string | null,
164                 target: SqlOperandType,
165                 As: string | null = null,
166                 alias: string | null = null) {
167         this.relation = relation
168         this.target = target
169         this.as = As || null
170         this.alias = alias || null
171     }
172 }
173
174 export class SqlJoin {
175     readonly type = JOIN_TYPE
176     joinType: JOINSTRING
177     condition: SqlConditional | ['using', TargetList] | null
178     lhs: SqlJoin | SqlRelation
179     rhs: SqlJoin | SqlRelation
180
181     constructor(lhs: SqlJoin | SqlRelation,
182                rhs: SqlJoin | SqlRelation,
183                joinType: JOINSTRING = 'join',
184                condition: SqlConditional | ['using', TargetList] | null = null
185            ) {
186         this.lhs = lhs
187         this.rhs = rhs
188         this.joinType = joinType || 'join'
189         this.condition = condition || null
190     }
191 }
192
193 export class SqlRelation {
194     readonly type = RELATION_TYPE
195     target: SqlRelation | SqlJoin | string
196     alias: string | null
197
198     constructor(target: SqlRelation | SqlJoin | string,
199                alias: string | null = null) {
200         this.target = target
201         this.alias = alias || null
202     }
203 }
204
205 export type SqlConditionalOp = 'or' | 'and' | 'not' | 'in' | 'exists' | 'like'
206     |
207     | 'between' | 'isnull' | '<>' | 'contains' |
208     | '<=' | '>=' | '=' | '<' | '>' | '!='
209
210 export class SqlConditional {
211     readonly type = CONDITIONAL_TYPE
212     operation: SqlConditionalOp
213     lhs: SqlConditional | SqlOperandType
214     rhs: SqlConditional | SqlOperandType | null
215     not: boolean
216
217     constructor(operation: SqlConditionalOp,
218                lhs: SqlConditional | SqlOperandType,
219                rhs: SqlConditional | SqlOperandType | null = null,
220                not: boolean = false) {
221         if (operation === 'in' && lhs instanceof Array && lhs.length === 1)

```

```

221     lhs = lhs[0]
222     this.operation = operation
223     this.lhs = lhs
224     this.rhs = rhs || null
225     this.not = not
226   }
227 }
228
229 export type AggFuncName = 'avg' | 'count' | 'max' | 'min' | 'sum'
230
231 export class SqlAggFunction {
232   readonly type = AGGFUNCTION_TYPE
233   fname: AggFuncName
234   expr: SqlOperandType | TargetClause
235
236   constructor(fname: AggFuncName, expr: SqlOperandType | TargetClause) {
237     this.fname = fname
238     this.expr = expr
239   }
240 }
241
242 export type SqlOperationOps = '||' | '+' | '-' | '*' | '/'
243
244 export class SqlOperation {
245   readonly type = OPERATION_TYPE
246   op: SqlOperationOps
247   lhs: SqlOperandType
248   rhs: SqlOperandType
249
250   constructor(op: SqlOperationOps, lhs: SqlOperandType, rhs: SqlOperandType) {
251     this.op = op
252     this.lhs = lhs
253     this.rhs = rhs
254   }
255 }
256
257 /** RELATIONAL ALGEBRA */
258 // literals are strings
259
260 export type RelRelationish = RelRelation | RelJoin
261 export type RelOperandType = RelOperation | string | RelColumn
262
263 export class RelOperation {
264   readonly type = REL_OPERATION_TYPE
265   op: SqlOperationOps | 'union' | 'intersect' | 'except'
266   lhs: RelOperandType | HighLevelRelationish
267   rhs: RelOperandType | HighLevelRelationish
268
269   constructor(op: SqlOperationOps | 'union' | 'intersect' | 'except',
270     lhs: RelOperandType | HighLevelRelationish,
271     rhs: RelOperandType | HighLevelRelationish) {
272     this.op = op
273     this.lhs = lhs
274     this.rhs = rhs
275   }
276 }
277
278 type ColumnValueType = Column | RelFunction | string
279

```

```

280 export class RelColumn {
281   readonly type = REL_COLUMN_TYPE
282   relation: RelRelation | null
283   target: ColumnValueType
284   as: string | null
285
286   constructor(relation: RelRelation | null,
287               target: ColumnValueType,
288               As: string | null = null) {
289     this.relation = relation
290     this.target = target
291     this.as = As || null
292   }
293
294   alias(alias?: string) {
295     if (!alias)
296       return this
297     return new RelColumn(this.relation, this.target, alias)
298   }
299 }
300
301 export class RelFunction {
302   readonly type = REL_FUNCTION_TYPE
303   fname: AggFuncName
304   expr: '*' | RelColumn // TODO: support correct args
305
306   constructor(fname: AggFuncName, expr: '*' | RelColumn) {
307     this.fname = fname
308     this.expr = expr
309   }
310 }
311
312 export type ThetaOp = 'eq' | 'neq' | 'leq' | 'geq' | '<' | '>' | 'and' | 'or' |
313                     'in'
314
315 export class RelConditional {
316   readonly type = REL_CONDITIONAL_TYPE
317   operation: ThetaOp
318   lhs: RelOperandType | RelConditional
319   rhs: RelOperandType | RelConditional | RelOperandType[]
320
321   constructor(op: ThetaOp, lhs: RelOperandType | RelConditional,
322               rhs: RelOperandType | RelConditional | RelOperandType[]) {
323     this.operation = op
324     this.lhs = lhs
325     this.rhs = rhs
326   }
327 }
328
329 export type HighLevelRelationish = RelRelationish | RelRestriction |
330                                   RelProjection | RelRename | RelOperation
331
332 export class RelRestriction {
333   readonly type = REL_RESTRICTION_TYPE
334   conditions: RelConditional
335   args: HighLevelRelationish
336
337   constructor(conditions: RelConditional, args: HighLevelRelationish) {
338     this.conditions = conditions

```

```

338     this.args = args
339   }
340 }
341
342 export class RelProjection {
343   readonly type = REL_PROJECTION_TYPE
344   columns: RelColumn[]
345   args: HighLevelRelationish
346
347   constructor(columns: RelColumn[], args: HighLevelRelationish) {
348     this.columns = columns
349     this.args = args
350   }
351 }
352
353 type _RelRenameInputType = RelRelation | RelColumn | RelFunction |
354                             RelRename | string
355
356 export class RelRename {
357   readonly type = REL_RENAME_TYPE
358   input: _RelRenameInputType
359   output: string
360   args: HighLevelRelationish
361
362   constructor(input: _RelRenameInputType,
363               output: string,
364               args: HighLevelRelationish) {
365     this.input = input
366     this.output = output
367     this.args = args
368   }
369 }
370
371 export class RelRelation {
372   readonly type = REL_RELATION_TYPE
373   name: string
374
375   constructor(name: string) {
376     this.name = name
377   }
378 }
379
380 export type RelJoinCond = "cross" | "left" | "right" | RelConditional
381
382 // cross
383 // natural (no condition)
384 // theta join (with condition)
385 // semi (left and right)
386 export class RelJoin {
387   readonly type = REL_JOIN_TYPE
388   lhs: HighLevelRelationish
389   rhs: HighLevelRelationish
390   condition: RelJoinCond
391
392   constructor(lhs: HighLevelRelationish,
393               rhs: HighLevelRelationish,
394               cond: RelJoinCond) {
395     this.lhs = lhs
396     this.rhs = rhs

```

```

397     this.condition = cond
398   }
399 }

```

## 2.4 src/parser/peg

### 2.4.1 src/parser/peg/sql.pegjs

```

1  /*
2   Initially inspired by grammar of the "Phoenix" SQL layer
3   (https://forcedotcom.github.io/phoenix/index.html)
4
5   Primarily based on PostgreSQL syntax:
6   https://www.postgresql.org/docs/9/static/sql-syntax.html
7   https://www.postgresql.org/docs/9/static/sql-select.html
8   https://github.com/postgres/postgres/blob/master/src/backend/parser/gram.y
9  */
10
11  start
12    = Statements
13
14  Statements
15    = _ lhs:Statement rhs:( _ ";" _ Statement )* _ ";"?
16    { return rhs.reduce((result, element) => result.concat(element[3]), [lhs]) }
17
18  Statement
19    = Selectish
20
21  Selectish
22    = SelectPair
23    / Select
24
25
26  SelectPair
27    = lhs:Select __
28    pairing:$ ( "UNION"i / "INTERSECT"i / "EXCEPT"i ) __
29    spec:( "ALL"i __ / "DISTINCT"i __ )?
30    rhs:( Selectish )
31    {
32      return new SqlSelectPair(pairing.toLowerCase(),
33                               spec && spec[0].toLowerCase(),
34                               lhs,
35                               rhs)
36    }
37
38  Select
39    = "SELECT"i __ what:TargetClause __
40    "FROM"i __ from:FromClause
41    where:( __ "WHERE"i __ WhereClause )?
42    groupBy:( __ "GROUP"i __ "BY"i __ GroupByClause )?
43    having:( __ "HAVING"i __ HavingClause )?
44    orderBy:( __ "ORDER"i __ "BY"i __ OrderByClause )?
45    {
46      return new SqlSelect(what, from, where && where[3], groupBy && groupBy[5],
47                           having && having[3], orderBy && orderBy[5])
48    }
49  / "(" _ sel:Select _ ")" { return sel }

```

```

50
51 TargetClause
52   = spec:$( "DISTINCT"i __ / "ALL"i __ )?
53   target:(
54     "*"
55     / TargetList
56   )
57   { return {
58     'type': TARGETCLAUSE_TYPE,
59     'specifier': spec ? spec.toLowerCase() : null,
60     'targetlist': target
61   }
62   }
63
64 FromClause
65   = from:RelationList
66
67 WhereClause
68   = where:Condition
69
70 GroupByClause
71   = groupBy:TargetList
72
73 HavingClause
74   = having:Condition
75
76 OrderByClause
77   = lhs:Ordering rhs:( _ "," _ Ordering )*
78   { return rhs.reduce((result, element) => result.concat(element[3]), [lhs]) }
79
80 Ordering
81   = expr:Operand
82   cond:(
83     -- "ASC"i { return 'asc' }
84     / -- "DESC"i { return 'desc' }
85     / -- "USING"i _ op:$( "<" / ">" ) { return op }
86   )?
87
88 RelationList
89   = item1:RelationItem _ "," _ items:RelationList
90   { return new SqlJoin(item1, items) }
91   / Join
92   / RelationItem
93
94 RelationItem "RelationItem"
95   = item:RelationThing __ ( "AS"i __ )? alias:Name
96   { return new SqlRelation(item, alias) }
97   / RelationThing
98
99 RelationThing
100  = "(" _ list:RelationList _ ")"
101  { return list }
102  / "(" _ join:Join _ ")"
103  { return join }
104  / tableName:Name
105  { return new SqlRelation(tableName) }
106
107 Join
108   = item1:RelationItem __

```



```

109     jtype:JoinType --
110     item2:RelationItem
111     jcond:(
112         -- "ON"i
113         -- expr:Condition
114         { return expr }
115         / -- "USING"i -
116         "(" _ list:TargetList _ ")"
117         { return ['using', list] }
118     )?
119     { return new SqlJoin(item1, item2, jtype, jcond) }
120
121 TargetList
122 = item1:TargetItem _ "," _ items:TargetList
123 { return [item1].concat(items) }
124 / item:TargetItem
125 { return [item] }
126
127 TargetItem "TargetItem"
128 = table:Name ".*"
129 { return new SqlColumn(table, '*', `${table}.*`, null) }
130 / op:Operand -- "AS"i -- alias:Name
131 { return new SqlColumn(null, op, alias, alias) }
132 / op:Operand -- alias:Name
133 { return new SqlColumn(null, op, alias, alias) }
134 / op:Operand _ "=" _ alias:Name
135 { return new SqlColumn(null, op, alias, alias) }
136 / op:Operand
137 { return (op instanceof SqlColumn) ? op : new SqlColumn(null, op) }
138
139 Condition "Condition"
140 = lhs:AndCondition rhs:( -- "OR"i -- Condition )?
141 { return rhs ? new SqlConditional('or', lhs, rhs[3]) : lhs }
142
143 AndCondition
144 = lhs:InnerCondition rhs:( -- "AND"i -- AndCondition )?
145 { return rhs ? new SqlConditional('and', lhs, rhs[3]) : lhs }
146
147 InnerCondition
148 = ( ConditionContains
149     / ConditionComp
150     / ConditionIn
151     / ConditionExists
152     / ConditionLike
153     / ConditionBetween
154     / ConditionNull
155     // / Operand
156 )
157 / "NOT"i -- expr:Condition
158 { return new SqlConditional('not', expr) }
159 / "(" _ expr:Condition _ ")"
160 { return expr }
161
162 ConditionContains "Conditional-Contains"
163 // based on Transact-SQL
164 = "CONTAINS" _
165 "(" _
166     lhs:(
167         Operand

```

```

168         / "(" _ ops:OperandList _ ")"
169         { return ops }
170     )
171     rhs:SQStringLiteral
172     ")"
173     { return new SqlConditional('contains', lhs, rhs) }
174
175 ConditionComp "Conditional-Comparison"
176 = lhs:Operand _ cmp:Compare _ rhs:Operand
177 { return new SqlConditional(cmp, lhs, rhs) }
178
179 ConditionIn
180 = lhs_op:Operand __
181   not:( "NOT"i __ )?
182   "IN"i _
183   "(" _
184   rhs_ops:( Selectish / OperandList ) _
185   ")"
186 { return new SqlConditional('in', lhs_op, rhs_ops, not) }
187
188 ConditionExists
189 = "EXISTS"i _
190   "(" _ subquery:Selectish _ ")"
191 { return new SqlConditional('exists', subquery) }
192
193 ConditionLike
194 = lhs_op:Operand __
195   not:( "NOT"i __ )?
196   "LIKE"i __
197   rhs_op:Operand
198 { return new SqlConditional('like', lhs_op, rhs_op, not) }
199
200 ConditionBetween
201 = lhs_op:Operand __
202   not:( "NOT"i __ )?
203   "BETWEEN"i
204   rhs:(
205     __
206     rhs_op1:Operand __
207     "AND"i __
208     rhs_op2:Operand
209     { return [rhs_op1, rhs_op2] }
210   / _
211   "(" _
212     rhs_op1:Operand __
213     "AND"i __
214     rhs_op2:Operand
215     ")"
216     { return [rhs_op1, rhs_op2] }
217   )
218 { return new SqlConditional('between', lhs_op, rhs, not) }
219
220 ConditionNull
221 = lhs:Operand __ "IS"i __
222   not:( "NOT"i __ )?
223   NullLiteral
224 { return new SqlConditional('isnull', lhs, null, not) }
225
226 Term

```

```

227     = Literal
228     / AggFunction
229     / "(" _ op:Operand _ ")" { return op }
230     / ColumnRef
231
232 ColumnRef
233     = tbl:( table:Name "." )? column:Name
234     { return new SqlColumn(tbl && tbl[0],
235         column,
236         tbl ? `${tbl[0]}.${column}` : column
237         ) }
238
239 AggFunction "aggregate function"
240     = AggFunctionAvg
241     / AggFunctionCount
242     / AggFunctionMax
243     / AggFunctionMin
244     / AggFunctionSum
245
246 AggFunctionAvg
247     = "AVG"i _
248     "(" _ term:Term _ ")"
249     { return new SqlAggFunction("avg", term) }
250
251 AggFunctionCount
252     = "COUNT"i _
253     "(" _
254     targ:TargetClause _
255     ")"
256     { return new SqlAggFunction("count", targ) }
257
258 AggFunctionMax
259     = "MAX"i _
260     "(" _
261     term:Term _
262     ")"
263     { return new SqlAggFunction("max", term) }
264
265 AggFunctionMin
266     = "MIN"i _
267     "(" _
268     term:Term _
269     ")"
270     { return new SqlAggFunction("min", term) }
271
272 AggFunctionSum
273     = "SUM"i _
274     "(" _
275     term:Term _
276     ")"
277     { return new SqlAggFunction("sum", term) }
278
279 /***** PRIMITIVES *****/
280
281 Name
282     = DQStringLiteral
283     / BQStringLiteral
284     / !ReservedWord id:Ident {return id }
285

```

```

286 Ident "UnquotedIdent"
287   = $( [A-Za-z_] [A-Za-z0-9_]* )
288
289 OperandList
290   = lhs:Operand
291     rhs:( _ "," _ Operand ) *
292     {
293       if (rhs.length)
294         return rhs.reduce((result, element) => result.concat(element[3]), [lhs])
295       else
296         return lhs
297     }
298
299 Operand // Summand | makeOperation
300   = lhs:Summand
301     rhs:( _ "||" _ Summand ) *
302     { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation("||",
303                                                                    lh, rh[3])) }
304   / Selectish
305
306 Summand // Factor | makeOperation
307   = lhs:Factor
308     rhs:( _ ("+" / "-") _ Factor ) *
309     { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation(rh[1],
310                                                                    lh, rh[3])) }
311
312 Factor // literal | function | Operand | column | makeOperation
313   = lhs:Term
314     rhs:( _ ("*" / "/" ) _ Term ) *
315     { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation(rh[1],
316                                                                    lh, rh[3])) }
317
318 Compare
319   = "<>"
320     / "<="
321     / ">="
322     / "="
323     / "<"
324     / ">"
325     / "!="
326
327 JoinType "JoinType"
328   = ( "CROSS"i __ )? "JOIN"i
329     { return "join" }
330   / "INNER"i __ "JOIN"i
331     { return "equi" }
332   / "NATURAL"i __ "JOIN"i
333     { return "natural" }
334   / "LEFT"i __ ( "OUTER"i __ )? "JOIN"i
335     { return "left" }
336   / "RIGHT"i __ ( "OUTER"i __ )? "JOIN"i
337     { return "right" }
338   / "FULL"i __ ( "OUTER"i __ )? "JOIN"i
339     { return "full" }
340
341 /***** LITERALS *****/
342
343 Literal "Literal"
344   = SQStringLiteral

```

```

345 / NumericLiteral
346 / ExponentialLiteral
347 / BooleanLiteral
348 / NullLiteral
349
350 BQStringLiteral "backtick string"
351 = $( ` ` ( [^`] / ```` )+ `` )
352
353 DQStringLiteral "double-quote string"
354 = $( "" ( [^"] / """" )+ "" )
355
356 SQStringLiteral "single-quote string"
357 = lit:$( "'" ( [^'] / "'" ) * "'" !SQStringLiteral )
358 { return new SqlLiteral('string', lit.slice(1, -1)) }
359 / lit:$( ("âĀĲ"/"âĀĴ") ( [^âĀĴ] ) * "âĀĴ" ) // fancy single-quote
360 { return new SqlLiteral('string', lit.slice(1, -1)) }
361
362 ExponentialLiteral "exponential"
363 = val:$( NumericLiteral "e" IntegerLiteral )
364 { return new SqlLiteral('number', parseFloat(val)) }
365
366 NumericLiteral "number"
367 = IntegerLiteral
368 / DecimalLiteral
369
370 IntegerLiteral "integer"
371 = int:$( "-"? [0-9]+ )
372 { return new SqlLiteral('number', parseInt(int)) }
373
374 DecimalLiteral "decimal"
375 = value:$( "-"? [0-9]+ "." [0-9]+ )
376 { return new SqlLiteral('number', parseFloat(value)) }
377
378 NullLiteral "null"
379 = "NULL"i
380 { return new SqlLiteral('null', null) }
381
382 BooleanLiteral "boolean"
383 = TruePrim
384 / FalsePrim
385
386 TruePrim
387 = "TRUE"i
388 { return new SqlLiteral('boolean', true) }
389
390 FalsePrim
391 = "FALSE"i
392 { return new SqlLiteral('boolean', false) }
393
394 - "OptWhitespace"
395 = WS* (Comment WS*)* {}
396
397 -- "ReqWhitespace"
398 = WS+ (Comment WS*)* {}
399
400 WS
401 = [ \t\n]
402
403 Comment "Comment"

```

```

404 = "/" * ( ! "*" / " . ) * "*" / " {}
405 / "--" ( !"n" . ) * "\n" {}
406
407 /** SQL2008 reserved words.
408     In alphabetical order but not always lexical order,
409     as there is no backtracking in PEG.js, e.g. for
410     "IN" / "INT" / "INTERSECT" / "INTERSECTION"
411     only "IN" is reachable.
412 **/
413 ReservedWord
414 = $(
415     "ABS"i / "ALL"i / "ALLOCATE"i / "ALTER"i / "AND"i / "ANY"i / "ARE"i /
416     "ARRAY_AGG"i / "ARRAY"i / "ASENSITIVE"i / "ASYMMETRIC"i / "AS"i /
417     "ATOMIC"i / "AT"i / "AUTHORIZATION"i / "AVG"i
418     / "BEGIN"i / "BETWEEN"i / "BIGINT"i / "BINARY"i / "BLOB"i / "BOOLEAN"i /
419     "BOTH"i / "BY"i
420     / "CALLED"i / "CALL"i / "CARDINALITY"i / "CASCADED"i / "CASE"i / "CAST"i
421     /
422     "CEILING"i / "CEIL"i / "CHARACTER_LENGTH"i / "CHAR_LENGTH"i /
423     "CHARACTER"i / "CHAR"i / "CHECK"i / "CLOB"i / "CLOSE"i / "COALESCE"i /
424     "COLLATE"i / "COLLECT"i / "COLUMN"i / "COMMIT"i / "CONDITION"i /
425     "CONNECT"i / "CONSTRAINT"i / "CONVERT"i / "CORRESPONDING"i / "CORR"i /
426     "COUNT"i / "COVAR_POP"i / "COVAR_SAMP"i / "CREATE"i / "CROSS"i /
427     "CUBE"i / "CUME_DIST"i / "CURRENT_CATALOG"i / "CURRENT_DATE"i /
428     "CURRENT_DEFAULT_TRANSFORM_GROUP"i / "CURRENT_PATH"i / "CURRENT_ROLE"i
429     /
430     "CURRENT_SCHEMA"i / "CURRENT_TIMESTAMP"i / "CURRENT_TIME"i /
431     "CURRENT_TRANSFORM_GROUP_FOR_TYPE"i / "CURRENT_USER"i / "CURRENT"i /
432     "CURSOR"i / "CYCLE"i
433     / "DATALINK"i / "DATE"i / "DAY"i / "DEALLOCATE"i / "DECIMAL"i /
434     "DECLARE"i / "DEC"i / "DEFAULT"i / "DELETE"i / "DENSE_RANK"i /
435     "DEREF"i / "DESCRIBE"i / "DETERMINISTIC"i / "DISCONNECT"i /
436     "DISTINCT"i / "DLNEWCOPY"i / "DLPREVIOUSCOPY"i / "DLURLCOMPLETE"i /
437     "DLURLCOMPLETEONLY"i / "DLURLCOMPLETEWRITE"i / "DLURLPATHONLY"i /
438     "DLURLPATHWRITE"i / "DLURLPATH"i / "DLURLSCHEME"i / "DLURLSERVER"i /
439     "DLVALUE"i / "DOUBLE"i / "DROP"i / "DYNAMIC"i
440     / "EACH"i / "ELEMENT"i / "ELSE"i / "END-EXEC"i / "END"i / "ESCAPE"i /
441     "EVERY"i / "EXCEPT"i / "EXECUTE"i / "EXEC"i / "EXISTS"i / "EXP"i /
442     "EXTERNAL"i / "EXTRACT"i
443     / "FALSE"i / "FETCH"i / "FILTER"i / "FIRST_VALUE"i / "FLOAT"i / "FLOOR"i
444     /
445     "FOREIGN"i / "FOR"i / "FREE"i / "FROM"i / "FULL"i / "FUNCTION"i /
446     "FUSION"i
447     / "GET"i / "GLOBAL"i / "GRANT"i / "GROUPING"i / "GROUP"i
448     / "HAVING"i / "HOLD"i / "HOUR"i
449     / "IDENTITY"i / "IMPORT"i / "INDICATOR"i / "INNER"i / "INOUT"i /
450     "INSENSITIVE"i / "INSERT"i / "INTEGER"i / "INTERSECTION"i /
451     "INTERSECT"i / "INTERVAL"i / "INTO"i / "INT"i / "IN"i / "IS"i
452     / "JOIN"i
453     / "LAG"i / "LANGUAGE"i / "LARGE"i / "LAST_VALUE"i / "LATERAL"i /
454     "LEADING"i / "LEAD"i / "LEFT"i / "LIKE_REGEX"i / "LIKE"i / "LN"i /
455     "LOCALTIMESTAMP"i / "LOCAL"i / "LOCALTIME"i / "LOWER"i
456     / "MATCH"i / "MAX_CARDINALITY"i / "MAX"i / "MEMBER"i / "MERGE"i /
457     "METHOD"i / "MINUTE"i / "MIN"i / "MODIFIES"i / "MODULE"i / "MOD"i /
458     "MONTH"i / "MULTISET"i
459     / "NATIONAL"i / "NATURAL"i / "NCHAR"i / "NCLOB"i / "NEW"i / "NONE"i /
460     "NORMALIZE"i / "NOT"i / "NO"i / "NTH_VALUE"i / "NTILE"i / "NULLIF"i /
461     "NULL"i / "NUMERIC"i
462     / "OCCURRENCES_REGEX"i / "OCTET_LENGTH"i / "OFFSET"i / "OF"i / "OLD"i /
463     "ONLY"i / "ON"i / "OPEN"i / "ORDER"i / "OR"i / "OUTER"i / "OUT"i /

```

```

460      "OVERLAPS"i / "OVERLAY"i / "OVER"i
461      / "PARAMETER"i / "PARTITION"i / "PERCENTILE_CONT"i / "PERCENTILE_DISC"i /
462      "PERCENT_RANK"i / "POSITION_REGEX"i / "POSITION"i / "POWER"i /
463      "PRECISION"i / "PREPARE"i / "PRIMARY"i / "PROCEDURE"i
464      / "RANGE"i / "RANK"i / "READS"i / "REAL"i / "RECURSIVE"i / "REFERENCES"i
465      /
466      "REFERENCING"i / "REF"i / "REGR_AVGX"i / "REGR_AVGY"i / "REGR_COUNT"i /
467      "REGR_INTERCEPT"i / "REGR_R2"i / "REGR_SLOPE"i / "REGR_SXX"i /
468      "REGR_SXY"i / "REGR_SYY"i / "RELEASE"i / "RESULT"i / "RETURNS"i /
469      "RETURN"i / "REVOKE"i / "RIGHT"i / "ROLLBACK"i / "ROLLUP"i / "ROWS"i /
470      "ROW_NUMBER"i / "ROW"i
471      / "SAVEPOINT"i / "SCOPE"i / "SCROLL"i / "SEARCH"i / "SECOND"i /
472      "SELECT"i / "SENSITIVE"i / "SESSION_USER"i / "SET"i / "SIMILAR"i /
473      "SMALLINT"i / "SOME"i / "SPECIFICTYPE"i / "SPECIFIC"i /
474      "SQLEXCEPTION"i / "SQLSTATE"i / "SQLWARNING"i / "SQL"i / "SQRT"i /
475      "START"i / "STATIC"i / "STDDEV_POP"i / "STDDEV_SAMP"i / "SUBMULTISET"i
476      /
477      "SUBSTRING_REGEX"i / "SUBSTRING"i / "SUM"i / "SYMMETRIC"i /
478      "SYSTEM_USER"i / "SYSTEM"i
479      / "TABLESAMPLE"i / "TABLE"i / "THEN"i / "TIMESTAMP"i / "TIMEZONE_HOUR"i /
480      "TIMEZONE_MINUTE"i / "TIME"i / "TO"i / "TRAILING"i /
481      "TRANSLATE_REGEX"i / "TRANSLATE"i / "TRANSLATION"i / "TREAT"i /
482      "TRIGGER"i / "TRIM_ARRAY"i / "TRIM"i / "TRUE"i / "TRUNCATE"i
483      / "UESCAPE"i / "UNION"i / "UNIQUE"i / "UNKNOWN"i / "UNNEST"i / "UPDATE"i
484      /
485      "UPPER"i / "USER"i / "USING"i
486      / "VALUES"i / "VALUE"i / "VARBINARY"i / "VARCHAR"i / "VARYING"i /
487      "VAR_POP"i / "VAR_SAMP"i
488      / "WHENEVER"i / "WHEN"i / "WHERE"i / "WIDTH_BUCKET"i / "WINDOW"i /
489      "WITHIN"i / "WITHOUT"i / "WITH"i
490      / "XMLAGG"i / "XMLATTRIBUTES"i / "XMLBINARY"i / "XMLCAST"i /
491      "XMLCOMMENT"i / "XMLCONCAT"i / "XMLDOCUMENT"i / "XMLELEMENT"i /
492      "XML EXISTS"i / "XMLFOREST"i / "XMLITERATE"i / "XMLNAMESPACES"i /
493      "XMLPARSE"i / "XMLPI"i / "XMLQUERY"i / "XMLSERIALIZE"i / "XMLTABLE"i /
494      "XMLTEXT"i / "XMLVALIDATE"i / "XML"i
495      / "YEAR"i
496  ) !Ident

```

## 2.4.2 src/parser/peg/relations.pegjs

```

1  |
2  start
3  = _ rel:Relations _
4  { return rel }
5
6  Relations
7  = lhs:Relation
8    rhs:( _ Relations )*
9  { return rhs.reduce((l, r) => l.concat(r[1]), [lhs]) }
10
11 Relation
12 = table:Name
13   - "(" -
14     cols:Columns
15   - ")" -
16 { return [table, cols] }
17

```

```

18 Columns
19   = lhs:Column rhs:( _ ", " _ Column )*
20   { return rhs.reduce((l,r) => l.concat([r[3]]), [lhs]) }
21
22 Column
23   = name:Name _ ":" _ typ:Ident
24   { return [name, typ] }
25
26
27 /* sql primitives */
28
29 Name "Name"
30   = DQStringLiteral
31     / BQStringLiteral
32     / Ident
33
34 Ident "UnquotedIdent"
35   = $( [A-Za-z_] [A-Za-z0-9_] * )
36
37 BQStringLiteral "backtick string"
38   = $( ` ` ( [^`] / ```` )+ ```` )
39
40 DQStringLiteral "double-quote string"
41   = $( " " ( [^"] / """" )+ """" )
42
43 - "OptWhitespace"
44   = WS* Comment? WS* {}
45
46 -- "ReqWhitespace"
47   = WS+ Comment? WS* {}
48   / WS* Comment? WS+ {}
49
50 WS
51   = [ \t\n]
52
53 Comment "Comment"
54   = "/" * ( !"*/" . ) * "*/" {}
55   / "--" ( !"\n" . ) * "\n" {}

```

## 2.5 src/query\_tree

### 2.5.1 src/query\_tree/node.ts

```

1 import {Operation} from './operation'
2
3 export default
4 class Node {
5   operation: Operation
6   children: Node[] = []
7
8   constructor(operation: Operation) {
9     this.operation = operation
10  }
11
12   addNode(node: Node) {
13     this.children.push(node)
14  }

```



15 | }

**2.5.2** src/query\_tree/operation.ts

```

1  |
2  | export class Operation {
3  |     name: string
4  |     //TODO: better define this type
5  |     arguments: string[] = []
6  |
7  |     constructor(name: string) {
8  |         this.name = name
9  |     }
10 |
11 |     addArgument(arg: string) {
12 |         this.arguments.push(arg)
13 |     }
14 | }
15 |
16 | export class Projection extends Operation {
17 |     constructor() {
18 |         super("Project")
19 |     }
20 |
21 |     addTarget(data) {
22 |         let { relation, target, alias } = data
23 |         let arg: string = `${relation} ${target}`
24 |         if(alias)
25 |             arg += ` as ${alias}`
26 |
27 |         this.addArgument(arg)
28 |     }
29 | }
30 |
31 | export class From extends Operation {
32 |     constructor() {
33 |         super("From")
34 |     }
35 |
36 |     addTarget(data) {
37 |         if(data.lhs && data.rhs) {
38 |             this.addTarget(data.lhs)
39 |             this.addTarget(data.rhs)
40 |             return
41 |         }
42 |
43 |         else if(data.lhs || data.rhs) {
44 |             throw new Error('From without both lhs and rhs')
45 |         }
46 |
47 |         let arg = data.target
48 |         if(data.alias) arg += ` as ${data.alias}`
49 |         this.addArgument(arg)
50 |     }
51 | }
52 |
53 | export class Where extends Operation {

```

```

54 |   constructor() {
55 |     super("Where")
56 |   }
57 |
58 |   addTarget(data) {
59 |     let lhs = this.getArgument(data.lhs)
60 |     let rhs = this.getArgument(data.rhs)
61 |     this.addArgument(lhs + ` ${data.operation} ` + rhs)
62 |   }
63 |
64 |   getArgument(data): string {
65 |     if(data.lhs && data.rhs) {
66 |       let lhs = this.getArgument(data.lhs)
67 |       let rhs = this.getArgument(data.rhs)
68 |       let arg = lhs + ` ${data.operation} ` + rhs
69 |       return arg
70 |     } else if(data.lhs || data.rhs) {
71 |       throw new Error('lhs and rhs not both specified')
72 |     }
73 |
74 |     let arg
75 |     if(data.relation) arg = `${data.relation}.${data.target}`
76 |     if(data.relation && data.alias) arg += ` as ${data.alias}`
77 |     if(data.value) arg = data.value
78 |
79 |     return arg
80 |   }
81 | }

```

### 2.5.3 src/query\_tree/parse.ts

```

1 | // tslint:disable
2 | import Node from './node'
3 | import {Operation, Projection, From, Where} from './operation'
4 |
5 | // convert json produced by peg to Tree
6 | export default function parseSQLToTree(sql): Node {
7 |   // TODO: fix order of tree hierarchy
8 |
9 |   let projectArgs = sql[0].what.targetlist
10 |   let op = new Projection()
11 |   projectArgs.forEach(arg => op.addTarget(arg))
12 |   let root = new Node(op)
13 |
14 |   let fromArgs = sql[0].from
15 |   let from = new From()
16 |   from.addTarget(fromArgs)
17 |   let fromNode = new Node(from)
18 |   root.addNode(fromNode)
19 |
20 |   let whereArgs = sql[0].where
21 |   let where = new Where()
22 |   where.addTarget(whereArgs)
23 |   let whereNode = new Node(where)
24 |   fromNode.addNode(whereNode)
25 |
26 |   return root

```

27 | } 