CS 5300 Project #1

Jared Rainwater & Samuel K. Grush

December 5, 2017

Contents

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 `Statement`s. 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 seperate `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 `TargetItem`s. 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 $^1/_3$ 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`.

1.3 Optimization

Once a query has been processed, it is ready for execution. But performance gains are possible by further analyzing the quey and restructuring it. This is an optimization process that works by looking at the structure and operations of the query and modifying it based on specific rules and heuristics. All of the optimization are based on specific transformations.

1.3.1 Transformation Rules

These transformations are based on those described in Elmasri on page 698–699.

1. **cascading of restrictions:** If a restriction is made that has multiple conditions anded together, an individual restrict can be made and they can be nested within one another.
2. **commute restriction:** If a restrictions operates directly on another restriction nested within it, then the order of the restrictions can be changed.
3. **cascade selection:** In a sequence of selection operations, all but the last one can be ignored.
4. **commute restriction with selection:** If a selection deals only with attributes pertaining to selection, the restriction and selection are commutative.
5. **commute cross product or join:** The relations associated with either the cross product or join can have their order changed.
6. **commute selection with join/cross product:** A selection on the result of a join or cross product can move to just an individual relation if its conditions only apply to attributes from the relation.
7. **Commute restriction with join/cross product:** A restriction that operates on the result of a join or cross product can be split into 2 restrictions operating on either argument. Either restrictions' columns will now pertain only to the relation it acts on.
8. **commute set operations:** union and intersection are commutative.
9. **associativity of join, cross product, union and intersection:** the mentioned operations are associative with themselves
10. **selection commutes with set operations:** for any set operation, a selection operating on its result is the same as it operating on the arguments
11. **restriction commutes with union:** a restriction on a union is the same as restricting the arguments of the union
12. **convert selection on a cross product to join:** When using a selection on a cross product, you can convert it to a join with its condition matching the selection.
13. **selection in conjuction with set minus:** If commuting a selection with set minus, the select only moves to the first, left hand argument.
14. **selection in conjuction with intersect:** if a selection acting on an intersection has conditions pertaining only to 1 of the arguments, you can apply the selection to just that argument

The following optimizations or based on page 700

1. transformation 1 can be used to break select statements into cascades, allowing for more freedom in applying other optimizations
2. Rules pertaining to the commutativity of selection can be used to push them as far down the tree as possible. This can be a leaf node if it deals with one relation or a join if it deals with more.

3. Rules 5 and 9 are used to execute the most restrictive selections first to reduce the amount of processing as early as possible
4. Use Rule 12 wherever possible to convert cross products to joins
5. With rules 3, 4, 7 and 11 you can move restrictions as far down the tree as possible
6. Identify any groups of operations that can be implemented with a single operation

2 Source Code

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

2.1 src/

2.1.1 src/index.ts

```
import * as React from "react"
import * as ReactDOM from "react-dom"

import './styles/tests.scss'

import Main from './Main'

import {involves} from './parser/relAnalysis'
import {selectResults} from './parser/tests'
Object.assign(window, {
  involves,
  selectResults,
  mainResult: [null, null]
})

ReactDOM.render(
  React.createElement(Main),
  document.getElementById("content")
)
```

2.1.2 src/Main.tsx

```
import * as React from "react"

import * as JSONPretty from 'react-json-pretty'

const Tracer = require('pegjs-backtrace')

import {Catalog} from './parser/types'

import RelationsInput, {RelationsInputOutput} from './components/RelationsInput
    '
import QueryInput from './components/QueryInput'
import Tests from './components/Tests'
import TestCase from './components/TestCase'

export interface MainState {
  queryInputText: string
  status: string
  queryJSON: any
  relJSON: any
  catalog: Catalog.Catalog | null

  debug: string
}

export default class Main extends React.Component<any, MainState> {

  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    onRelationsInputUpdate(output: RelationsInputOutput) {
42      if (output.error) {
43        this.setState({
44          catalog: null,
45          status: `Error Parsing Relations:  ${output.error}`,
46          debug: output.traceback
47        })
48      } else {
49        this.setState({
50          catalog: output.catalog,
51          status: "Successfully Parsed Relations",
52          debug: ''
53        })
54      }
55    }
56
57    onQueryInputUpdate(text: string): void {
58      this.setState({
59        status: "Parsing Query...",
60        queryInputText: text,
61        queryJSON: null,
62        relJSON: null,
63        debug: ""
64      })
65
66    }
67
68    render() {
69      return (
70        <main id="main">
71          <RelationsInput onUpdate={this.onRelationsInputUpdate} />
72          <QueryInput
73            onUpdate={this.onQueryInputUpdate}
74            disabled={!this.state.catalog}
75          />
76          <div id="parse-status">{this.state.status}</div>
77          <div id="main-output">
78            <TestCase
79              catalog={this.state.catalog}
80              queryInputText={this.state.queryInputText}
81              doRun={true} // bad idea??
82              anchor="main-test"
83              resultTuple={(window as any).mainResult}
84              name="Main Test"
85              showStructures={undefined}
```

```
          />
          <div id="debug-output" data-empty={!this.state.debug}>
            <pre><code>{this.state.debug}</code></pre>
          </div>
        </div>
        <hr />
        <hr />
        <Tests catalog={this.state.catalog} />
      </main>
    )
  }
}
```

2.2 src/components

2.2.1 src/components/QueryInput.tsx

```
import * as React from "react"

export interface QueryInputProps {
  onUpdate: (text: string) => void
  disabled: boolean
}

export default class QueryInput extends React.Component<QueryInputProps, any> {
  textInput: HTMLTextAreaElement

  constructor(props: QueryInputProps) {
    super(props)

    this.onSubmit = this.onSubmit.bind(this)
  }

  onSubmit(e?) {
    if (e) e.preventDefault()
    console.info("Submitting:", this.textInput.value)
    this.props.onUpdate(this.textInput.value)
  }
  render() {
    return (
      <div id="query-input-wrapper">
        <textarea
          id="query-input"
          placeholder="Query..."
          cols={80}
          rows={10}
          ref={(input: HTMLTextAreaElement) => {this.textInput = input}}
        />
        <button
          disabled={this.props.disabled}
          onClick={this.onSubmit}
        >Parse Query</button>
      </div>
    )
  }
}
```

2.2.2 src/components/RelationsInput.tsx

```tsx
import * as React from "react"

const Tracer = require('pegjs-backtrace')

import {parseRelations} from '../parser/parsing'
import {Catalog} from '../parser/types'

const DEFAULT_INPUT = `
Sailors(sid:integer, sname:string, rating:integer, age:real)
Boats(bid:integer, bname:string, color:string)
Reserves(sid:integer, bid:integer, day:date)
`

export interface RelationsInputOutput {
  catalog: Catalog.Catalog | null
  error: null | Error
  traceback: '' | string
}

export interface RelationsInputProps {
  onUpdate: (output: RelationsInputOutput) => void
}

interface RelationsInputState {
  catalog: Catalog.Catalog | null
  text: string
}

export default class RelationsInput extends React.Component<RelationsInputProps
    , RelationsInputState> {

  constructor(props) {
    super(props)
    this.state = {
      catalog: null,
      text: DEFAULT_INPUT
    }

    this.run = this.run.bind(this)
    this.onChange = this.onChange.bind(this)
  }

  run(e?) {
    const text = this.state.text
    if (e) e.preventDefault()

    const tracer = new Tracer(text, {
      useColor: false,
      showTrace: true
    })

    let catalog: Catalog.Catalog|null = null
    try {
      catalog = parseRelations(text, {tracer})
      this.props.onUpdate({ catalog, error: null, traceback: '' })
```

```
    } catch (ex) {
      this.props.onUpdate({
        catalog,
        error: ex,
        traceback: tracer.getParseTreeString()
      })
    }
    this.setState({catalog})
  }

  onChange(event) {
    this.setState({text: event.target.value})
  }

  render() {
    return (
      <div id="relations-input-wrapper">
        <textarea
          id="relations-input"
          value={this.state.text}
          cols={80}
          rows={10}
          onChange={this.onChange}
        />
        <button onClick={this.run}>Parse Relations</button>
      </div>
    )
  }
}
```

2.2.3 src/components/TestCase.tsx

```
import * as React from "react"
import * as JSONPretty from 'react-json-pretty'
const Tracer = require('pegjs-backtrace')

import {ResultTuple} from '../parser/tests'
import {Catalog} from '../parser/types'
import {parseSql, SqlSyntaxError, sqlToRelationalAlgebra} from '../parser/
    parsing'
import {htmlHLR} from '../parser/relationalText'

import {Projection} from '../query_tree/operation'
import Node from '../query_tree/node'
import Tree from '../components/tree'

interface TestCaseProps {
  catalog: Catalog.Catalog | null
  queryInputText: string
  doRun: boolean
  anchor: string
  resultTuple: ResultTuple
  showStructures: boolean | undefined
  name?: string
}

interface TestCaseState {
```

```
25    status: string
26    treeStatus: string
27    queryJSON: any
28    relAlJSON: any
29    root: Node | null
30    relAlHTML: JSX.Element | null
31    color: string
32    tscolor: string
33    debug: any
34    showStructures: boolean
35  }
36
37  export default class TestCase extends React.Component<TestCaseProps,
        TestCaseState> {
38
39    constructor(props) {
40      super(props)
41      this.state = this.initialState()
42
43      this.run = this.run.bind(this)
44      this.toggleStructures = this.toggleStructures.bind(this)
45    }
46
47    componentDidMount() {
48      this.propsReceived(this.props)
49    }
50
51    componentWillReceiveProps(newProps: TestCaseProps) {
52      this.propsReceived(newProps)
53    }
54
55    propsReceived(newProps: TestCaseProps) {
56      const {catalog, queryInputText, doRun, showStructures} = this.props
57      if (newProps.showStructures !== undefined)
58        this.setState({showStructures: newProps.showStructures})
59
60      // if any test-related props are different, reset state.
61      if (newProps.catalog !== catalog ||
62          newProps.queryInputText !== queryInputText ||
63          newProps.doRun !== doRun
64         ) {
65        this.setState(this.initialState(), () => {
66          if (newProps.catalog && newProps.queryInputText && newProps.doRun)
67            this.run(newProps)
68        })
69      }
70    }
71
72    initialState(): TestCaseState {
73      return {
74        status: 'init',
75        treeStatus: '',
76        queryJSON: null,
77        relAlJSON: null,
78        relAlHTML: null,
79        root: null,
80        color: 'currentcolor',
81        tscolor: 'currentcolor',
82        debug: '',
```

```
83        showStructures: Boolean(this.props.showStructures)
84      }
85    }
86
87    run(props: TestCaseProps = this.props) {
88
89      const catalog = props.catalog as Catalog.Catalog
90
91      const tracer = new Tracer(props.queryInputText, {
92        useColor: false,
93        showTrace: true
94      })
95
96      let status
97      let treeStatus = ''
98      let queryJSON
99      let relAlJSON
100     let relAlHTML
101     let root: Node | null = null
102     let color = 'currentcolor'
103     let tscolor = 'currentcolor'
104     let debug = ''
105
106     try {
107       queryJSON = parseSql(props.queryInputText, {tracer})
108       this.props.resultTuple[0] = queryJSON
109       status = "SQL Scanned and Tokenized"
110       color = "green"
111     } catch (ex) {
112       if (ex instanceof SqlSyntaxError)
113         status = `Parser Syntax Error: ${ex.message}`
114       else
115         status = `Other Parser ${ex}`
116       console.error(ex)
117       color = "red"
118       debug = tracer.getParseTreeString()
119     }
120
121     if (queryJSON) {
122       try {
123         relAlJSON = sqlToRelationalAlgebra(queryJSON, catalog)
124         this.props.resultTuple[1] = relAlJSON
125         status = "SQL Parsed and converted to Relational Algebra"
126         color = "green"
127       } catch (ex) {
128         this.props.resultTuple[0] = ex
129         status = `Relational Algebra ${ex}`
130         color = "red"
131         console.error(ex)
132       }
133     }
134     if (relAlJSON) {
135       try {
136         relAlHTML = htmlHLR(relAlJSON)
137         status = "Relational Algebra rendered to HTML"
138         color = "green"
139       } catch (ex) {
140         this.props.resultTuple[1] = ex
141         status = `HTML Conversion Error: ${ex}`
```

```
142         color = "red"
143         console.error(ex)
144       }
145       if (relAlJSON)
146         try {
147           root = new Node(relAlJSON)
148           status = "Tree Generated"
149           color = "green"
150         } catch (ex) {
151           treeStatus = `Tree Error: ${ex}`
152           tscolor = "red"
153           console.error(ex)
154         }
155     }
156
157     this.setState({
158       status,
159       treeStatus,
160       queryJSON,
161       relAlJSON,
162       relAlHTML,
163       root,
164       color,
165       tscolor,
166       debug
167     })
168   }
169
170   toggleStructures(e) {
171     this.setState({showStructures: !this.state.showStructures})
172   }
173
174   render() {
175     return (
176       <section id={this.props.anchor} className="testcase">
177         <hr />
178         <h3>{this.props.name || this.props.anchor}</h3>
179         <pre><code>{this.props.queryInputText}</code></pre>
180         <div className="testcase-status">
181           <span style={{color: this.state.color}}>
182             Status: {this.state.status || "OK"}
183           </span>
184           { this.state.treeStatus && (
185             <span style={{color: this.state.tscolor}}>
186               Tree Status: {this.state.treeStatus}
187             </span>
188           )}
189         </div>
190         { (this.state.queryJSON || this.state.relAlJSON) && (
191             <button onClick={this.toggleStructures}>
192               {this.state.showStructures ? "Hide" : "Show"} Structures
193             </button>
194           )
195         }
196         <div className="testcase-inner">
197           <div className="relal-html" data-empty={!this.state.relAlHTML}>
198             <h4>Relational Algebra</h4>
199             {this.state.relAlHTML}
200           </div>
```

```
201             <div
202               className="sql-json"
203               data-empty={!(this.state.queryJSON && this.state.showStructures)}
204             >
205               <h4>SQL Structure</h4>
206               <JSONPretty json={this.state.queryJSON} />
207             </div>
208             <div
209               className="relal-json"
210               data-empty={!(this.state.relAlJSON && this.state.showStructures)}
211             >
212               <h4>Relational Algebra Structure</h4>
213               <JSONPretty json={this.state.relAlJSON} />
214             </div>
215             <div className="tree" data-empty={!this.state.root}>
216               <h4>Tree</h4>
217               { this.state.root &&
218                   <Tree root={this.state.root} margin={10} />
219               }
220             </div>
221             <div className="traceback" data-empty={!this.state.debug}>
222               <h4>Error Traceback</h4>
223               <pre><code>{this.state.debug}</code></pre>
224             </div>
225           </div>
226         </section>
227       )
228     }
229   }
```

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, selectResults} 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.Catalog | null
15 }
16
17 interface TestsState {
18   catalog: Catalog.Catalog | null
19   doRun: boolean
20   queryNames: string[]
21   showStructures: boolean | undefined
22 }
23
24 export default class Tests extends React.Component<TestsProps, TestsState> {
25   constructor(props) {
```

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

```
85              }
86            </ol>
87          </nav>
88          <div id="tests-list">
89            {
90              selectTests.map((testStr, idx) => (
91                <TestCase
92                  queryInputText={testStr}
93                  catalog={this.state.catalog}
94                  doRun={this.state.doRun}
95                  key={idx}
96                  anchor={`q${idx}`}
97                  resultTuple={selectResults[idx]}
98                  name={this.state.queryNames[idx] || undefined}
99                  showStructures={this.state.showStructures}
100               />
101             ))
102           }
103         </div>
104       </div>
105     )
106   }
107 }
```

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 export default
11 class Tree extends React.Component<TreeProps, any> {
12   render() {
13       const rows: JSX.Element[] = []
14       let frontier: Node[] = [this.props.root]
15       let key = 0
16       while (frontier.length > 0) {
17         const node: Node = frontier.shift() as Node
18         const row = <TreeRow
19                       node={node}
20                       key={key}
21                       offset={node.depth}/>
22         rows.push(row)
23         frontier = node.children.concat(frontier)
24         key++
25       }
26       return (
27       <div>
28         {rows}
29       </div>
30       )
31   }
```

```
}

interface TreeRowProps {
  offset: number
  node: Node
}

class TreeRow extends React.Component<TreeRowProps, any> {
  render() {
    return (
      <div className="tree-row">
        {"-".repeat(this.props.offset) + `${this.props.offset})`} {this.props.
            node.operation.html}
      </div>
    )
  }
}
```

2.3 src/parser

2.3.1 src/parser/parsing.ts

```

import { parse as RelationParse } from './peg/relations'
import { parse as SqlParse } from './peg/sql'
export { SyntaxError as SqlSyntaxError } from './peg/sql'
import {Sql, Rel, Catalog} from './types'
import {fromSqlSelect, fromSelectPair} from './sqlToRel'

export function parseRelations(input: string, args?): Catalog.Catalog {
  return Catalog.Catalog.fromParse(RelationParse(input, args))
}

export function parseSql(input: string, args?) {
  return SqlParse(input, args)
}

export function sqlToRelationalAlgebra(sqlStatements, catalog: Catalog.Catalog)
    {
  if (!Array.isArray(sqlStatements))
    throw new Error("Expected SQL statements")
  if (sqlStatements.length > 1)
    throw new Error("Multiple statements not supported")

  const TLStatement = sqlStatements[0]
  if (TLStatement instanceof Sql.Select)
    return fromSqlSelect(TLStatement, catalog)
  else if (TLStatement instanceof Sql.SelectPair)
    return fromSelectPair(TLStatement, catalog)
  else
    throw new Error(`Unknown sqlToRelationalAlgebra arg ${TLStatement}`)
}
```

2.3.2 src/parser/relAnalysis.ts

```
1
2  import {Rel, Catalog} from './types'
3
4  type RelationSet = Set<Catalog.Relation>
5  type ColumnSet = Set<Catalog.Column>
6  type InvolvementTuple = [RelationSet, ColumnSet]
7
8  type IterableTuple<T = any, U = T> = [Iterable<T>, Iterable<U>]
9
10 export function isJoinCondition(cond: Rel.Conditional,
11                                  left: InvolvementTuple,
12                                  right: InvolvementTuple) {
13   if (typeof cond.lhs === 'string' || typeof cond.rhs === 'string' ||
14       cond.operation === 'or' || cond.operation === 'and' ||
15       cond.operation === 'in' || Array.isArray(cond.rhs))
16     return false
17
18   const condLhs = involves(cond.lhs)[0]
19   const condRhs = involves(cond.rhs)[0]
20
21   const condLhs_left_exclusive = new Set()
22   const condLhs_right_exclusive = new Set()
23   const condRhs_left_exclusive = new Set()
24   const condRhs_right_exclusive = new Set()
25   condLhs.forEach((rel) => {
26     if (left[0].has(rel) && !right[0].has(rel) && !condRhs.has(rel))
27       condLhs_left_exclusive.add(rel)
28     else if (right[0].has(rel) && !left[0].has(rel) && !condLhs.has(rel))
29       condLhs_right_exclusive.add(rel)
30   })
31   condRhs.forEach((rel) => {
32     if (left[0].has(rel) && !right[0].has(rel) && !condLhs.has(rel))
33       condRhs_left_exclusive.add(rel)
34     else if (right[0].has(rel) && !left[0].has(rel) && !condRhs.has(rel))
35       condRhs_right_exclusive.add(rel)
36   })
37
38   if (condLhs_left_exclusive.size && condRhs_right_exclusive.size &&
39       !condLhs_right_exclusive.size && !condRhs_left_exclusive.size)
40     return 'join'
41   if (!condLhs_left_exclusive.size && !condRhs_right_exclusive.size &&
42       condLhs_right_exclusive.size && condRhs_left_exclusive.size)
43     return 'join-swap'
44
45   return false
46 }
47
48 /**
49  * Set([A, B]) = _union([A], [B])
50  *
51  * let x = new Set([A])
52  * _union(x, Set([B]))
53  * x == Set([A, B])
54  */
55 function _union<T>(base: Iterable<T> | null,
56                    ...args: Array<Iterable<T>>): Set<T> {
57   let newSet: Set<T>
58   if (!base)
59     newSet = new Set<T>()
```

```
60    else if (!(base instanceof Set))
61      newSet = new Set<T>(base)
62    else
63      newSet = base
64
65    for (const arg of args) {
66      for (const b of arg) {
67        newSet.add(b)
68      }
69    }
70    return newSet
71  }
72
73  function _unionZip(base: InvolvementTuple, arg: IterableTuple): void
74  function _unionZip<T, U>(base: [Set<T>, Set<U>], arg: IterableTuple<T, U>):
        void {
75    _union(base[0], arg[0])
76    _union(base[1], arg[1])
77  }
78
79  function newInvolvementTuple(relations: Iterable<Catalog.Relation> = [],
80                               columns: Iterable<Catalog.Column> = []
81    ): InvolvementTuple {
82    return [
83      new Set(relations),
84      new Set(columns)
85    ]
86  }
87
88  type Involvable = Rel.HighLevelRelationish | Rel.Column | Rel.Conditional |
89                    Rel.RelFunction
90
91  export function involves(involved: Involvable): InvolvementTuple {
92    if (involved instanceof Rel.HLR)
93      switch (involved.type) {
94        case Rel.HLRTypeString.Aggregation:
95          return involves_Aggregation(involved as Rel.Aggregation)
96        case Rel.HLRTypeString.Restriction:
97          return involves_Restriction(involved as Rel.Restriction)
98        case Rel.HLRTypeString.Projection:
99          return involves_Projection(involved as Rel.Projection)
100       case Rel.HLRTypeString.Rename:
101         return involves_Rename(involved as Rel.Rename)
102       case Rel.HLRTypeString.Relation:
103         return involves_Relation(involved as Rel.Relation)
104       case Rel.HLRTypeString.Join:
105         return involves_Join(involved as Rel.Join)
106       case Rel.HLRTypeString.Operation:
107         return involves_Operation(involved as Rel.Operation)
108
109       default:
110         console.info("Unexpected HLR", involved.type)
111         throw new Error(`Unexpected Rel.HLRTypeString "${involved.type}"`)
112     }
113   else if (involved instanceof Rel.Column)
114     return involves_Column(involved)
115   else if (involved instanceof Rel.Conditional)
116     return involves_Conditional(involved)
117   else if (involved instanceof Rel.RelFunction) {
```

```
118     // TODO: is this one needed?
119     throw new Error("involves_Function() not implemented")
120   } else {
121     console.error("involved:", involved)
122     throw new Error("Unexpected type to involves()")
123   }
124 }
125
126 function involves_Operand(operand: Rel.OperandType|Rel.HighLevelRelationish
127   ): null|InvolvementTuple {
128   if (typeof operand === 'string')
129     return null
130   else if (operand instanceof Rel.Column)
131     return involves_Column(operand)
132   else if (operand instanceof Rel.HLR) {
133     return involves(operand as Rel.HighLevelRelationish)
134   }
135   console.error("involves_Operand", operand)
136   throw new Error("Unexpected argument to involves_Operand")
137 }
138
139 function involves_Operation(op: Rel.Operation): InvolvementTuple {
140   const invTuple = newInvolvementTuple()
141
142   const lhsInv = involves_Operand(op.lhs)
143   const rhsInv = involves_Operand(op.rhs)
144
145   if (lhsInv) _unionZip(invTuple, lhsInv)
146   if (rhsInv) _unionZip(invTuple, rhsInv)
147
148   return invTuple
149 }
150
151 function involves_Relation(relation: Rel.Relation): InvolvementTuple {
152   const invTuple = newInvolvementTuple([relation.target])
153   // TODO: should this add all of the relation's columns too??
154   return invTuple
155 }
156
157 function involves_Function(funct: Rel.RelFunction): InvolvementTuple {
158   const invTuple = newInvolvementTuple()
159   const {fname, expr} = funct
160
161   if (expr === '*') {
162     if (!funct.hlr)
163       throw new Error("RelFunction of '*' with no hlr-hint")
164     _unionZip(invTuple, involves(funct.hlr))
165   } else if (expr instanceof Rel.Column)
166     _unionZip(invTuple, involves_Column(expr))
167   else {
168     console.error("involves_Function", funct)
169     throw new Error("Unexpected argument to involves_Function")
170   }
171
172   return invTuple
173 }
174
175 function involves_Column(col: Rel.Column): InvolvementTuple {
176   const invTuple = newInvolvementTuple()
```

```
177   const target = col.target
178
179   if (col.relation)
180     invTuple[0].add(col.relation.target)
181
182   if (target instanceof Catalog.Column)
183     invTuple[1].add(target)
184   else if (target instanceof Rel.RelFunction) {
185     involves_Function(target)
186   } else if (typeof target === 'string') {
187     // don't do anything
188   } else
189     throw new Error("Unexpected argument to involves_Column")
190
191   return invTuple
192 }
193
194 function involves_Join(join: Rel.Join): InvolvementTuple {
195   const invTuple = newInvolvementTuple()
196
197   const lhsInv = involves_Operand(join.lhs)
198   const rhsInv = involves_Operand(join.rhs)
199   const conInv = join.condition instanceof Rel.Conditional
200                  ? involves_Conditional(join.condition)
201                  : null
202
203   if (lhsInv) _unionZip(invTuple, lhsInv)
204   if (rhsInv) _unionZip(invTuple, rhsInv)
205   if (conInv) _unionZip(invTuple, conInv)
206
207   return invTuple
208 }
209
210 function involves_Conditional(conditional: Rel.Conditional): InvolvementTuple {
211   const invTuple = newInvolvementTuple()
212
213   const {lhs, rhs} = conditional
214
215   if (lhs instanceof Rel.Conditional)
216     _unionZip(invTuple, involves_Conditional(lhs))
217   else if (lhs instanceof Rel.RelFunction)
218     _unionZip(invTuple, involves_Function(lhs))
219   else {
220     const inv = involves_Operand(lhs)
221     if (inv) _unionZip(invTuple, inv)
222   }
223
224   if (rhs instanceof Rel.Conditional)
225     _unionZip(invTuple, involves_Conditional(rhs))
226   else if (rhs instanceof Rel.RelFunction)
227     _unionZip(invTuple, involves_Function(rhs))
228   else if (Array.isArray(rhs))
229     for (const operand of rhs) {
230       const inv = involves_Operand(operand)
231       if (inv) _unionZip(invTuple, inv)
232     }
233   else {
234     const inv = involves_Operand(rhs)
235     if (inv) _unionZip(invTuple, inv)
```

```
236   }
237
238   return invTuple
239 }
240
241 function involves_Restriction(restriction: Rel.Restriction): InvolvementTuple {
242   const invTuple = newInvolvementTuple()
243
244   _unionZip(invTuple, involves_Conditional(restriction.conditions))
245   _unionZip(invTuple, involves(restriction.args))
246
247   return invTuple
248 }
249
250 function involves_Projection(projection: Rel.Projection): InvolvementTuple {
251   const invTuple = newInvolvementTuple()
252
253   projection.columns.forEach((col) => {
254     if (col instanceof Rel.Column)
255       _unionZip(invTuple, involves_Column(col))
256     else {
257       // TODO: col is a literal; should we do anything?
258     }
259   })
260
261   // TODO: should the Column involvement be reset?
262   const invArgs = involves(projection.args)
263   // if (column involvement should be reset)
264   //   invArgs[1].clear()
265   _unionZip(invTuple, invArgs)
266
267   return invTuple
268 }
269
270 function involves_Rename(ren: Rel.Rename): InvolvementTuple {
271   // const invTuple = newInvolvementTuple()
272   // TODO: Do we really need to do anything with ren.input?? Assuming not
273   return involves(ren.args)
274 }
275
276 function involves_Aggregation(agg: Rel.Aggregation): InvolvementTuple {
277   const invTuple = newInvolvementTuple()
278
279   agg.attributes.forEach((col) => _unionZip(invTuple, involves_Column(col)))
280   // TODO: do anything with agg.functions ?
281   _unionZip(invTuple, involves(agg.relation))
282
283   return invTuple
284 }
```

2.3.3 src/parser/relationalText.tsx

```
1 import * as React from 'react'
2
3 const ReactDOMServer = require('react-dom/server')
4
5 import {Rel, Sql, Catalog} from './types'
```

```
6
7  export function getSymbol(input: string) {
8    switch (input) {
9      // passthroughs
10     case '||':
11     case '+':
12     case '-':
13     case '*':
14     case '/':
15     case '<':
16     case '>':
17       return input
18
19     case 'aggregation':
20       return "âĎŚ" // U+2111
21     case 'restriction':
22       return "ÏČ"
23     case 'projection':
24       return "Îă"
25     case 'rename':
26       return "ÏĄ"
27     case 'rename-divider':
28       return "âĹŢ"
29
30     case 'union':
31       return "âĹŧ"
32     case 'intersect':
33       return "âĹľ"
34     case 'except':
35       return "âĹŠ"
36
37     case 'join':
38       return "âŃĹ"
39     case 'left':
40     case 'ljoin':
41       return "âŃĽ"
42     case 'right':
43     case 'rjoin':
44       return "âŃŁ"
45     case 'cross':
46     case 'crossjoin':
47       return "âĺĽ"
48     case 'divide':
49       return "Ãů"
50
51     case 'eq':
52       return "="
53     case 'neq':
54       return "âľă"
55     case 'leq':
56       return "âľđ"
57     case 'geq':
58       return "âľě"
59     case 'and':
60       return "âĹğ"
61     case 'or':
62       return "âĹĺ"
63     case 'in':
64       return "âĹŁ"
```

```
65      default:
66        throw new Error(`Unknown symbol name "${input}"`)
67    }
68  }
69
70  export function htmlARGS(args: Rel.HighLevelRelationish, noargs = false) {
71    if (noargs) {
72      return null
73    } else {
74      const ARGS = htmlHLR(args)
75      return (
76        <span className="args">
77          (
78            <span className="HLR">
79              {ARGS}
80            </span>
81          )
82        </span>
83      )
84    }
85  }
86
87  export function htmlRelAggregation(agg: Rel.Aggregation, noargs = false) {
88    let attrs: JSX.Element | null = null
89    if (agg.attributes && agg.attributes.length)
90      attrs = (
91        <sub className="columns">
92          (
93            {htmlColumnList(agg.attributes)}
94          )
95        </sub>
96      )
97
98    const aggregJsx = (
99      <span className="RelAggregation">
100       {attrs}
101       <span className="operator">{getSymbol('aggregation')}</span>
102       <sub className="functions">
103         {htmlColumnList(agg.functions)}
104       </sub>
105       {htmlARGS(agg.relation, noargs)}
106     </span>
107   )
108
109   if (agg.renames.length)
110     return htmlRelRenameAggregate(agg.renames, aggregJsx)
111   else
112     return aggregJsx
113 }
114
115 export function htmlRelRestriction(res: Rel.Restriction, noargs = false) {
116   const SYM = getSymbol('restriction')
117   const COND = htmlRelConditional(res.conditions)
118   const ARGS = htmlARGS(res.args, noargs)
119   return (
120     <span className="RelRestriction">
121       <span className="operator">{SYM}</span>
122       <sub className="condition">
123         {COND}
```

```
124         </sub>
125         {ARGS}
126       </span>
127     )
128   }
129
130   export function htmlRelProjection(res: Rel.Projection, noargs = false) {
131     const SYM = getSymbol('projection')
132     const ARGS = htmlARGS(res.args, noargs)
133     return (
134       <span className="RelProjection">
135         <span className="operator">{SYM}</span>
136         <sub className="columns">
137           {htmlColumnList(res.columns)}
138         </sub>
139         {ARGS}
140       </span>
141     )
142   }
143
144   export function htmlColumnList(cols: Array<string|Rel.Column|Rel.RelFunction>
145     ): Array<string|JSX.Element> {
146     const columns: Array<string|JSX.Element> = []
147     cols.forEach((col, idx) => {
148       if (idx > 0)
149         columns.push(",")
150       if (col instanceof Rel.Column)
151         columns.push(htmlRelColumn(col, idx))
152       else if (col instanceof Rel.RelFunction)
153         columns.push(htmlRelFunction(col, idx))
154       else
155         columns.push(col)
156     })
157     return columns
158   }
159
160   export function htmlRelColumn(col: Rel.Column, iter?: number) {
161
162     if (col.as) {
163       return (
164         <span className="RelColumn" key={iter}>
165           <span className="column-as">{col.as}</span>
166         </span>
167       )
168     }
169
170     if (!col.relation) {
171       return (
172         <span className="RelColumn" key={iter}>
173           <span className="column-name">{getName(col.target)}</span>
174         </span>
175       )
176     }
177
178     return (
179       <span className="RelColumn" key={iter}>
180         <span className="relation-name">{getName(col.relation)}</span>
181         .
182         <span className="column-name">{getName(col.target)}</span>
```

```
183       </span>
184     )
185   }
186
187   export function htmlRelFunction(funct: Rel.RelFunction, idx?) {
188     const NAME = funct.fname.toUpperCase()
189     const EXPR = funct.expr === '*'
190             ? '*'
191             : htmlRelColumn(funct.expr)
192
193     return (
194       <span className="RelFunction" key={idx}>
195         <span className="function-name">{NAME}</span>
196         (
197           {EXPR}
198         )
199       </span>
200     )
201   }
202
203   export function getName(thing) {
204     if (typeof(thing) === 'string')
205       return thing
206     if (thing instanceof Rel.Relation)
207       return thing.name
208     if (thing instanceof Rel.Column)
209       return thing.as || htmlRelColumn(thing)
210     if (thing instanceof Rel.RelFunction)
211       return htmlRelFunction(thing as Rel.RelFunction)
212     if (thing instanceof Catalog.Column)
213       return thing.name
214     console.info("getName", thing)
215     throw new Error("unexpected thing to getName")
216   }
217
218   export function htmlRelRenameAggregate(renames: string[], aggregJsx: JSX.
        Element) {
219     const SYM = getSymbol('rename')
220     const OUTPUT = htmlColumnList(renames)
221     return (
222       <span className="RelRename RelRename-aggregation">
223         <span className="operator">{SYM}</span>
224         <sub className="condition">
225           {OUTPUT}
226         </sub>
227         (
228           {aggregJsx}
229         )
230       </span>
231     )
232   }
233
234   export function htmlRelRename(ren: Rel.Rename, noargs = false) {
235     const SYM = getSymbol('rename')
236     const INPUT = getName(ren.input)
237     const OUTPUT = ren.output
238     const ARGS = htmlARGS(ren.args, noargs)
239
240     // R as S  =>  ÏĄ_S (R)
```

```
241   if (ren.input === ren.args && ren.input instanceof Rel.Relation)
242     return (
243       <span className="RelRename RelRename-unary">
244         <span className="operator">{SYM}</span>
245         <sub className="condition">
246           {OUTPUT}
247         </sub>
248         {ARGS}
249       </span>
250     )
251
252   // R.a as b  =>  ÏĄ_{bâĹŢR.a}(R)
253   // R as S  =>  ÏĄ_{SâĹŢR}(...)
254   return (
255     <span className="RelRename">
256       <span className="operator">{SYM}</span>
257       <sub className="condition">
258         {OUTPUT} {getSymbol('rename-divider')} {INPUT}
259       </sub>
260       {ARGS}
261     </span>
262   )
263 }
264
265 export function htmlRelRelation(rel: Rel.Relation) {
266   const NAME = rel.name
267   return (
268     <span className="RelRelation">
269       {NAME}
270     </span>
271   )
272 }
273
274 export function relJoinHelper(join: Rel.Join): [string, JSX.Element | null] {
275   if (typeof(join.condition) === 'string') {
276     return [getSymbol(join.condition), null]
277   } else if (join.condition instanceof Rel.Conditional) {
278     let cond = htmlRelConditional(join.condition)
279     if (cond) {
280       cond = (
281         <sub className="condition">
282           {cond}
283         </sub>
284       )
285     }
286     return [getSymbol('join'), cond]
287   } else {
288     throw new Error(`unknown RelJoin condition ${join.condition}`)
289   }
290 }
291
292 export function htmlRelJoin(join: Rel.Join) {
293   const [joinSymbol, cond] = relJoinHelper(join)
294   const LHS = htmlHLR(join.lhs)
295   const RHS = htmlHLR(join.rhs)
296
297   return (
298     <span className="RelJoin">
299       {LHS}
```

```
300         <span className="operator">{joinSymbol}</span>
301         {cond}
302         {RHS}
303       </span>
304     )
305   }
306
307   export function htmlRelOperation(op: Rel.Operation) {
308     const OPSYM = getSymbol(op.op)
309     const LHS = htmlRelOperand(op.lhs)
310     const RHS = htmlRelOperand(op.rhs)
311
312     return (
313       <span className="RelOperation">
314         {LHS}
315         <span className="operator">{OPSYM}</span>
316         {RHS}
317       </span>
318     )
319   }
320
321   export function htmlRelOperand(operand: Rel.OperandType|Rel.
        HighLevelRelationish) {
322     if (typeof(operand) === 'string')
323       return operand
324     if (operand instanceof Rel.RelFunction)
325       return htmlRelFunction(operand)
326     if (operand instanceof Rel.Operation)
327       return htmlRelOperation(operand)
328     if (operand instanceof Rel.Column)
329       return htmlRelColumn(operand)
330     // throw new Error("Unexpected operand type")
331     return htmlHLR(operand)
332   }
333
334   export function htmlRelConditional(cond: Rel.Conditional) {
335     const OPSYM = getSymbol(cond.operation)
336
337     let lhs
338     let rhs
339
340     if (cond.lhs instanceof Rel.Conditional)
341       lhs = htmlRelConditional(cond.lhs)
342     else if (cond.lhs instanceof Rel.RelFunction)
343       lhs = htmlRelFunction(cond.lhs)
344     else
345       lhs = htmlRelOperand(cond.lhs)
346
347     if (cond.rhs instanceof Rel.Conditional)
348       rhs = htmlRelConditional(cond.rhs)
349     else if (Array.isArray(cond.rhs))
350       rhs = cond.rhs.map(htmlRelOperand)
351     else if (cond.rhs instanceof Rel.RelFunction)
352       rhs = htmlRelFunction(cond.rhs)
353     else
354       rhs = htmlRelOperand(cond.rhs)
355
356     return (
357       <span className="RelConditional">
```

```
358       <span className="lhs">
359         {lhs}
360       </span>
361       <span className="operator">{OPSYM}</span>
362       <span className="rhs">
363         {rhs}
364       </span>
365     </span>
366   )
367 }
368
369 export function htmlHLR(hlr: Rel.HighLevelRelationish) {
370   if (hlr instanceof Rel.Restriction)
371     return htmlRelRestriction(hlr)
372   if (hlr instanceof Rel.Projection)
373     return htmlRelProjection(hlr)
374   if (hlr instanceof Rel.Rename)
375     return htmlRelRename(hlr)
376   if (hlr instanceof Rel.Operation)
377     return htmlRelOperation(hlr)
378   if (hlr instanceof Rel.Relation)
379     return htmlRelRelation(hlr)
380   if (hlr instanceof Rel.Join)
381     return htmlRelJoin(hlr)
382   if (hlr instanceof Rel.Aggregation)
383     return htmlRelAggregation(hlr)
384   console.error("unknown HLR:", hlr)
385   throw new Error("Unknown type passed to htmlHLR")
386 }
387
388 export function svgRelAggregation(agg: Rel.Aggregation) {
389   let attrs: JSX.Element | null = null
390   if (agg.attributes && agg.attributes.length)
391     attrs = (
392       <tspan baselineShift="sub" className="columns">
393         (
394           {svgColumnList(agg.attributes)}
395         )
396       </tspan>
397     )
398
399   const aggregJsx = (
400     <tspan className="RelAggregation">
401       {attrs}
402       <tspan className="operator">{getSymbol('aggregation')}</tspan>
403       <tspan baselineShift="sub" className="functions">
404         {svgColumnList(agg.functions)}
405       </tspan>
406     </tspan>
407   )
408
409   if (agg.renames.length)
410     return svgRelRenameAggregate(agg.renames, aggregJsx)
411   else
412     return aggregJsx
413 }
414
415 export function svgRelRestriction(res: Rel.Restriction) {
416   const SYM = getSymbol('restriction')
```

```
417   const COND = svgRelConditional(res.conditions)
418   return (
419     <tspan className="RelRestriction">
420       <tspan className="operator">{SYM}</tspan>
421       <tspan baselineShift="sub" className="condition">
422         {COND}
423       </tspan>
424     </tspan>
425   )
426 }
427
428 export function svgRelProjection(res: Rel.Projection) {
429   const SYM = getSymbol('projection')
430   return (
431     <tspan className="RelProjection">
432       <tspan className="operator">{SYM}</tspan>
433       <tspan baselineShift="sub" className="columns">
434         {svgColumnList(res.columns)}
435       </tspan>
436     </tspan>
437   )
438 }
439
440 export function svgColumnList(cols: Rel.Columnish[]
441   ): Array<string|JSX.Element> {
442   const columns: Array<string|JSX.Element> = []
443   cols.forEach((col, idx) => {
444     if (idx > 0)
445       columns.push(",")
446     if (col instanceof Rel.Column)
447       columns.push(svgRelColumn(col, idx))
448     else if (col instanceof Rel.RelFunction)
449       columns.push(svgRelFunction(col, idx))
450     else
451       columns.push(col)
452   })
453   return columns
454 }
455
456 export function svgRelColumn(col: Rel.Column, iter?: number) {
457
458   if (col.as) {
459     return (
460       <tspan className="RelColumn" key={iter}>
461         <tspan className="column-as">{col.as}</tspan>
462       </tspan>
463     )
464   }
465
466   if (!col.relation) {
467     return (
468       <tspan className="RelColumn" key={iter}>
469         <tspan className="column-name">{svgGetName(col.target)}</tspan>
470       </tspan>
471     )
472   }
473
474   return (
475     <tspan className="RelColumn" key={iter}>
```

```
476       <tspan className="relation-name">{svgGetName(col.relation)}</tspan>
477       .
478       <tspan className="column-name">{svgGetName(col.target)}</tspan>
479     </tspan>
480   )
481 }
482
483 export function svgRelFunction(funct: Rel.RelFunction, idx?) {
484   const NAME = funct.fname.toUpperCase()
485   const EXPR = funct.expr === '*'
486           ? '*'
487           : svgRelColumn(funct.expr)
488
489   return (
490     <tspan className="RelFunction" key={idx}>
491       <tspan className="function-name">{NAME}</tspan>
492       (
493         {EXPR}
494       )
495     </tspan>
496   )
497 }
498
499 export function svgGetName(thing) {
500   if (typeof(thing) === 'string')
501     return thing
502   if (thing instanceof Rel.Relation)
503     return thing.name
504   if (thing instanceof Rel.Column)
505     return thing.as || svgRelColumn(thing)
506   if (thing instanceof Rel.RelFunction)
507     return svgRelFunction(thing)
508   if (thing instanceof Catalog.Column)
509     return thing.name
510   console.info("svgGetName", thing)
511   throw new Error("unexpected thing to svgGetName")
512 }
513
514 export function svgRelRenameAggregate(renames: string[], aggregJsx: JSX.Element
      ) {
515   const SYM = getSymbol('rename')
516   const OUTPUT = svgColumnList(renames)
517   return (
518     <tspan className="RelRename RelRename-aggregation">
519       <tspan className="operator">{SYM}</tspan>
520       <tspan baselineShift="sub" className="condition">
521         {OUTPUT}
522       </tspan>
523       (
524         {aggregJsx}
525       )
526     </tspan>
527   )
528 }
529
530 export function svgRelRename(ren: Rel.Rename) {
531   const SYM = getSymbol('rename')
532   const INPUT = svgGetName(ren.input)
533   const OUTPUT = ren.output
```

```
534
535   // R as S  =>  ÏĄ_S (R)
536   if (ren.input === ren.args && ren.input instanceof Rel.Relation)
537     return (
538       <tspan className="RelRename RelRename-unary">
539         <tspan className="operator">{SYM}</tspan>
540         <tspan baselineShift="sub" className="condition">
541           {OUTPUT}
542         </tspan>
543       </tspan>
544     )
545
546   // R.a as b  =>  ÏĄ_{bâĹŢR.a}(R)
547   // R as S  =>  ÏĄ_{SâĹŢR}(...)
548   return (
549     <tspan className="RelRename">
550       <tspan className="operator">{SYM}</tspan>
551       <tspan baselineShift="sub" className="condition">
552         {OUTPUT} {getSymbol('rename-divider')} {INPUT}
553       </tspan>
554     </tspan>
555   )
556 }
557
558 export function svgRelRelation(rel: Rel.Relation) {
559   const NAME = rel.name
560   return (
561     <tspan className="RelRelation">
562       {NAME}
563     </tspan>
564   )
565 }
566
567 export function svgRelJoinHelper(join: Rel.Join): [string, JSX.Element | null]
      {
568   if (typeof(join.condition) === 'string') {
569     return [getSymbol(join.condition), null]
570   } else if (join.condition instanceof Rel.Conditional) {
571     let cond = svgRelConditional(join.condition)
572     if (cond) {
573       cond = (
574         <tspan baselineShift="sub" className="condition">
575           {cond}
576         </tspan>
577       )
578     }
579     return [getSymbol('join'), cond]
580   } else {
581     throw new Error(`unknown RelJoin condition ${join.condition}`)
582   }
583 }
584
585 export function svgRelJoin(join: Rel.Join) {
586   const [joinSymbol, cond] = svgRelJoinHelper(join)
587   const LHS = svgHLR(join.lhs)
588   const RHS = svgHLR(join.rhs)
589
590   return (
591     <tspan className="RelJoin">
```

```
592         {LHS}
593         <tspan className="operator">{joinSymbol}</tspan>
594         {cond}
595         {RHS}
596       </tspan>
597     )
598   }
599
600   export function svgRelOperation(op: Rel.Operation) {
601     const OPSYM = getSymbol(op.op)
602     const LHS = svgRelOperand(op.lhs)
603     const RHS = svgRelOperand(op.rhs)
604
605     return (
606       <tspan className="RelOperation">
607         {LHS}
608         <tspan className="operator">{OPSYM}</tspan>
609         {RHS}
610       </tspan>
611     )
612   }
613
614   export function svgRelOperand(operand: Rel.OperandType|Rel.HighLevelRelationish
        ) {
615     if (typeof(operand) === 'string')
616       return operand
617     if (operand instanceof Rel.RelFunction)
618       return svgRelFunction(operand)
619     if (operand instanceof Rel.Operation)
620       return svgRelOperation(operand)
621     if (operand instanceof Rel.Column)
622       return svgRelColumn(operand)
623     // throw new Error("Unexpected operand type")
624     return svgHLR(operand)
625   }
626
627   export function svgRelConditional(cond: Rel.Conditional) {
628     const OPSYM = getSymbol(cond.operation)
629
630     let lhs
631     let rhs
632
633     if (cond.lhs instanceof Rel.Conditional)
634       lhs = svgRelConditional(cond.lhs)
635     else if (cond.lhs instanceof Rel.RelFunction)
636       lhs = svgRelFunction(cond.lhs)
637     else
638       lhs = svgRelOperand(cond.lhs)
639
640     if (cond.rhs instanceof Rel.Conditional)
641       rhs = svgRelConditional(cond.rhs)
642     else if (Array.isArray(cond.rhs))
643       rhs = cond.rhs.map(svgRelOperand)
644     else if (cond.rhs instanceof Rel.RelFunction)
645       rhs = svgRelFunction(cond.rhs)
646     else
647       rhs = svgRelOperand(cond.rhs)
648
649     return (
```

```
650       <tspan className="RelConditional">
651         <tspan className="lhs">
652           {lhs}
653         </tspan>
654         <tspan className="operator">{OPSYM}</tspan>
655         <tspan className="rhs">
656           {rhs}
657         </tspan>
658       </tspan>
659     )
660   }
661
662   export function svgHLR(hlr: Rel.HighLevelRelationish) {
663     if (hlr instanceof Rel.Restriction)
664       return svgRelRestriction(hlr)
665     if (hlr instanceof Rel.Projection)
666       return svgRelProjection(hlr)
667     if (hlr instanceof Rel.Rename)
668       return svgRelRename(hlr)
669     if (hlr instanceof Rel.Operation)
670       return svgRelOperation(hlr)
671     if (hlr instanceof Rel.Relation)
672       return svgRelRelation(hlr)
673     if (hlr instanceof Rel.Join)
674       return svgRelJoin(hlr)
675     if (hlr instanceof Rel.Aggregation)
676       return svgRelAggregation(hlr)
677     console.error("unknown HLR:", hlr)
678     throw new Error("Unknown type passed to svgHLR")
679   }
680
681   export function getSVGString(hlr: Rel.HighLevelRelationish) {
682     const svg = (
683       <text className="svg-hlr">{svgHLR(hlr)}</text>
684     )
685
686     return ReactDOMServer.renderToStaticMarkup(svg)
687   }
688
689   (window as any).getSVGString = getSVGString
```

2.3.4 src/parser/relDupe.ts

```
1
2   import {Rel, Catalog, PairingString} from './types'
3
4   type Copiable = string | Catalog.Relation | Catalog.Column
5
6   type Dupable = Rel.HighLevelRelationish | Rel.Column | Rel.Conditional |
7                  Rel.RelFunction | Rel.PairingOperation | Copiable
8
9   export default function dupe(thing: Dupable) {
10    if (thing instanceof Rel.HLR)
11      switch (thing.type) {
12        case Rel.HLRTypeString.Aggregation:
13          return dupe_Aggregation(thing as Rel.Aggregation)
14        case Rel.HLRTypeString.Restriction:
```

```
        return dupe_Restriction(thing as Rel.Restriction)
      case Rel.HLRTypeString.Projection:
        return dupe_Projection(thing as Rel.Projection)
      case Rel.HLRTypeString.Rename:
        return dupe_Rename(thing as Rel.Rename)
      case Rel.HLRTypeString.Relation:
        return dupe_Relation(thing as Rel.Relation)
      case Rel.HLRTypeString.Join:
        return dupe_Join(thing as Rel.Join)
      case Rel.HLRTypeString.Operation:
        return dupe_Operation(thing as Rel.Operation)

      default:
        console.info("Unexpected HLR", thing.type)
        throw new Error(`Unexpected Rel.HLRTypeString "${thing.type}"`)
    }
  else if (thing instanceof Rel.Column)
    return dupe_Column(thing)
  else if (thing instanceof Rel.Conditional)
    return dupe_Conditional(thing)
  else if (thing instanceof Rel.RelFunction)
    return dupe_Function(thing)
  else if (typeof thing === 'string' ||
           thing instanceof Catalog.Column ||
           thing instanceof Catalog.Relation)
    return thing
  else
    throw new Error("Unexpected type to dupe()")
}

function dupe_Operation(op: Rel.Operation) {
  return new Rel.Operation(
    op.op,
    dupe(op.lhs),
    dupe(op.rhs)
  )
}

function dupe_Column(column: Rel.Column) {
  return new Rel.Column(
    column.relation && dupe(column.relation),
    dupe(column.target),
    column.as
  )
}

function dupe_Function(funct: Rel.RelFunction) {
  return new Rel.RelFunction(
    funct.fname,
    dupe(funct.expr),
    funct.hlr && dupe(funct.hlr)
  )
}

function dupe_Aggregation(agg: Rel.Aggregation) {
  return new Rel.Aggregation(
    agg.attributes.map(dupe),
    agg.functions.map(dupe),
    dupe(agg.relation),
```

```
    agg.renames.slice()
  )
}

function dupe_Conditional(cond: Rel.Conditional) {
  return new Rel.Conditional(
    cond.operation,
    dupe(cond.lhs),
    Array.isArray(cond.rhs) ? cond.rhs.map(dupe) : dupe(cond.rhs)
  )
}

function dupe_Restriction(restr: Rel.Restriction) {
  return new Rel.Restriction(
    dupe(restr.conditions),
    dupe(restr.args)
  )
}

function dupe_Projection(proj: Rel.Projection) {
  return new Rel.Projection(
    proj.columns.map(dupe),
    dupe(proj.args)
  )
}

function dupe_Rename(ren: Rel.Rename) {
  const input = dupe(ren.input)
  return new Rel.Rename(
    input,
    ren.output,
    ren.args === ren.input
      ? input as Rel.HighLevelRelationish
      : dupe(ren.args)
  )
}

function dupe_Relation(rel: Rel.Relation) {
  return new Rel.Relation(rel.name, rel.target)
}

function dupe_Join(join: Rel.Join) {
  return new Rel.Join(
    dupe(join.lhs),
    dupe(join.rhs),
    dupe(join.condition)
  )
}
```

2.3.5 src/parser/relTransform.ts

```

import {Rel, Catalog, PairingString} from './types'
import {involves, isJoinCondition} from './relAnalysis'
import dupe from './relDupe'

const PairingStrings: ReadonlyArray<PairingString>
```

```
  = ['union', 'intersect', 'except']

/**
 * arrayExtend(a, b)
 * Appends b elements to a, modifying a in place and returning it.
 */
function arrayExtend<U, V>(a: U[], b: V[]): Array<U|V> {
  Array.prototype.push.apply(a, b)
  return a
}

function inArray<U, V>(thing: U, array: ReadonlyArray<V>) {
  return (array as ReadonlyArray<U|V>).indexOf(thing) !== -1
}

function recursiveConditionSplit(cond: Rel.Conditional,
                                 op: 'and' | 'or'
  ) {

  if (cond.operation !== op || Array.isArray(cond.rhs))
    return [cond]

  const args: Rel.Conditional[] = []
  for (const hs of [cond.lhs, cond.rhs]) {
    if (!(hs instanceof Rel.Conditional)) {
      console.error("recursiveConditionSplit:", hs, cond, op)
      throw new Error("non-conditional parameter")
    }
    arrayExtend(args, recursiveConditionSplit(hs, op))
  }

  return args
}

/** Transformation Rule #1: Cascade of ÏČ */
function cascadeRestrictions(restr: Rel.Restriction, returnNew = false) {

  if (restr.conditions.operation !== 'and')
    return restr

  const conditions = recursiveConditionSplit(restr.conditions, 'and')
  const topCondition = conditions.pop()
  if (!topCondition) {
    console.error("cascadeRestrictions:", restr, conditions, topCondition)
    throw new Error("Unexpectedly empty conditions")
  }

  let newHLR = restr.args
  for (const cond of conditions) {
    newHLR = new Rel.Restriction(cond, newHLR)
  }

  if (returnNew)
    return new Rel.Restriction(dupe(topCondition), dupe(newHLR))

  return Object.assign(restr, {
    conditions: topCondition,
    args: newHLR
  })
```

```
66  }
67
68  /** Transformation Rule #1: Cascade of ÏČ (reverse) */
69  function rollupRestrictions(restr: Rel.Restriction, returnNew = false) {
70
71    // doesn't include restr.conditions
72    const conditionList: Rel.Conditional[] = []
73
74    let bottomHLR: Rel.HighLevelRelationish = restr.args
75    while (bottomHLR instanceof Rel.Restriction) {
76      conditionList.push(bottomHLR.conditions)
77      bottomHLR = bottomHLR.args
78    }
79
80    const newCondition = conditionList.reduce((accumulator, currentValue) => {
81      return new Rel.Conditional('and', accumulator, currentValue)
82    }, restr.conditions)
83
84    if (returnNew)
85      return new Rel.Restriction(dupe(newCondition), dupe(bottomHLR))
86
87    return Object.assign(restr, {
88      conditions: newCondition,
89      args: bottomHLR
90    })
91  }
92
93  /** Transformation Rule #2: Commutativity of ÏČ */
94  function commuteRestriction(restr: Rel.Restriction, returnNew = false) {
95
96    if (!(restr.args instanceof Rel.Restriction)) {
97      console.error("commuteRestriction:", restr)
98      throw new Error("Non-Restriction argument")
99    }
100
101   if (returnNew) {
102     const inner = new Rel.Restriction(restr.conditions, restr.args.args)
103     return dupe(new Rel.Restriction(restr.args.conditions, inner))
104   }
105
106   [restr.conditions, restr.args.conditions] =
107     [restr.args.conditions, restr.conditions]
108
109   return restr
110 }
111
112 /** Transformation Rule #3: Cascade of ÏĂ */
113 function condenseProjection(proj: Rel.Projection, returnNew = false) {
114
115   let bottomHLR: Rel.HighLevelRelationish = proj.args
116   while (bottomHLR instanceof Rel.Projection) {
117     bottomHLR = bottomHLR.args
118   }
119
120   if (returnNew)
121     return dupe(new Rel.Projection(proj.columns, bottomHLR))
122
123   return Object.assign(proj, {
124     args: bottomHLR
```

```
  })
}

function checkRestProjCommutativity(hlr: Rel.Restriction | Rel.Projection) {
  let condition: Rel.Conditional
  let columns: Array<string|Rel.Column>
  if (hlr instanceof Rel.Restriction) {
    if (!(hlr.args instanceof Rel.Projection)) {
      console.error("cRPC:", hlr, hlr.args)
      throw new Error("invalid Restriction argument")
    }
    condition = hlr.conditions
    columns = hlr.args.columns
  } else if (hlr instanceof Rel.Projection) {
    if (!(hlr.args instanceof Rel.Restriction)) {
      console.error("cRPC:", hlr, hlr.args)
      throw new Error("invalid Projection argument")
    }
    condition = hlr.args.conditions
    columns = hlr.columns
  } else {
    console.error("cRPC:", hlr)
    throw new Error("bad checkRestProjCommutativity argument type")
  }

  const cataColumns = new Set(
    columns.map((c: Rel.Column) => {
      if (typeof c.target === 'string')
        return null
      return c.target
    })
  )

  const [invRels, invCols] = involves(condition)
  for (const col of invCols) {
    if (!cataColumns.has(col))
      return false
  }
  return true
}

/** Transformation Rule #4: Commutating ÏČ with ÏĂ
 *  No way to perform destructively; always returns new without dupe.
 */
function commuteRestrictionProjection(hlr: Rel.Restriction | Rel.Projection) {
  const innerHLR = (hlr.args as Rel.Restriction | Rel.Projection).args

  if (hlr instanceof Rel.Restriction) {
    const columns = (hlr.args as Rel.Projection).columns
    const innerRestr = new Rel.Restriction(hlr.conditions, innerHLR)
    return new Rel.Projection(columns, innerRestr)
  } else if (hlr instanceof Rel.Projection) {
    const conds = (hlr.args as Rel.Restriction).conditions
    const innerProj = new Rel.Projection(hlr.columns, innerHLR)
    return new Rel.Restriction(conds, innerProj)
  } else {
    console.error("cRP:", hlr)
    throw new Error("bad commuteRestrictionProjection argument type")
  }
```

```
184 }
185
186 function checkJoinCommutativity(join: Rel.Join) {
187   return (join.condition instanceof Rel.Conditional ||
188       join.condition === "cross")
189 }
190
191 /** Transformation Rule #5: Commutativity of âŇĹ (and âĺĽ) */
192 function commuteJoin(join: Rel.Join, returnNew = false) {
193   if (returnNew)
194     return dupe(new Rel.Join(join.rhs, join.lhs, join.condition))
195   else
196     return Object.assign(join, {
197       lhs: join.rhs,
198       rhs: join.lhs
199     })
200 }
201
202 type restJoinCommType = 'lhs'|'rhs'|'split'|'split-swap'|boolean
203
204 function checkRestJoinCommutativity(restr: Rel.Restriction): restJoinCommType {
205   const args = restr.args as Rel.Join | Rel.PairingOperation
206   if (!(restr.args instanceof Rel.Join
207       || restr.args instanceof Rel.Operation))
208     return false
209
210   const condition = restr.conditions
211   const {lhs, rhs} = args
212
213   // TODO: make more efficient
214   const conditionInv = involves(condition)[1]
215   const lhsInv = involves(lhs)[1]
216   const rhsInv = involves(rhs)[1]
217
218   let condLhsInCommon = 0
219   let condRhsInCommon = 0
220   conditionInv.forEach((col) => {
221     if (lhsInv.has(col))
222       condLhsInCommon++
223     if (rhsInv.has(col))
224       condRhsInCommon++
225   })
226
227   if (!condLhsInCommon && !condRhsInCommon) {
228     console.log("What! Restriction unrelated to either arg???")
229     return true
230   } else if (!condRhsInCommon && condLhsInCommon === conditionInv.size)
231     return 'lhs'
232   else if (!condLhsInCommon && condRhsInCommon === conditionInv.size)
233     return 'rhs'
234
235   if (condition.operation !== 'and')
236     return false
237
238   const condLeftInv = involves(condition.lhs as Rel.Conditional)[1]
239   const condRightInv = involves(condition.rhs as Rel.Conditional)[1]
240
241   let lhsCondLeftInCommon = 0
242   let lhsCondRightInCommon = 0
```

```
243   let rhsCondLeftInCommon = 0
244   let rhsCondRightInCommon = 0
245   condLeftInv.forEach((col) => {
246     if (lhsInv.has(col))
247       lhsCondLeftInCommon++
248     if (rhsInv.has(col))
249       rhsCondLeftInCommon++
250   })
251   condRightInv.forEach((col) => {
252     if (lhsInv.has(col))
253       lhsCondRightInCommon++
254     if (rhsInv.has(col))
255       rhsCondRightInCommon++
256   })
257
258   if (!lhsCondRightInCommon && lhsCondLeftInCommon === condLeftInv.size &&
259       !rhsCondLeftInCommon && rhsCondRightInCommon === condRightInv.size)
260     return 'split'
261   if (!lhsCondLeftInCommon && lhsCondRightInCommon === condRightInv.size &&
262       !rhsCondRightInCommon && rhsCondLeftInCommon === condLeftInv.size)
263     return 'split-swap'
264
265   return false
266 }
267
268 /** Transformation Rule #6: Commuting ÏČ with âŃĹ (or âÍĽ)
269  *  No way to perform destructively; always returns new without dupe.
270  */
271 function commuteRestrictionJoin(restr: Rel.Restriction,
272                                 type: restJoinCommType) {
273   if (!type)
274     throw new Error("commuteRestrictionJoin on type = false")
275   if (type === true)
276     throw new Error("Ambiguous Commutativity")
277
278   const rCondition = restr.conditions
279   const rJoin = restr.args as Rel.Join
280
281   let newLhs
282   let newRhs
283
284   if (type === 'split' || type === 'split-swap') {
285     let newCondLhs
286     let newCondRhs
287     if (type === 'split') {
288       [newCondLhs, newCondRhs] = [rCondition.lhs, rCondition.rhs]
289     } else {
290       [newCondLhs, newCondRhs] = [rCondition.rhs, rCondition.lhs]
291     }
292     newLhs = new Rel.Restriction(newCondLhs, rJoin.lhs)
293     newRhs = new Rel.Restriction(newCondRhs, rJoin.rhs)
294   } else if (type === 'lhs') {
295     newLhs = new Rel.Restriction(rCondition, rJoin.lhs)
296     newRhs = rJoin.rhs
297   } else if (type === 'rhs') {
298     newLhs = rJoin.lhs
299     newRhs = new Rel.Restriction(rCondition, rJoin.rhs)
300   } else {
301     console.error("commuteRestrictionJoin:", restr, type)
```

```
302       throw new Error("Unexpected 'type' argument")
303     }
304
305     return new Rel.Join(newLhs, newRhs, rJoin.condition)
306   }
307
308   /** Transformation Rule #7: Commuting ÏĂ with âŃĹ (or âÍĽ) */
309   function commuteProjectionJoin(proj: Rel.Projection) {
310     if (!(proj.args instanceof Rel.Join))
311       throw new Error("Bad commuteProjectionJoin() argument")
312     const joinCond = proj.args.condition
313
314     const joinCondInv
315       = joinCond instanceof Rel.Conditional
316         ? involves(joinCond)[1]
317         : null
318
319     const lhsInv = involves(proj.args.lhs)[1]
320     const rhsInv = involves(proj.args.rhs)[1]
321
322     const projColumns = new Set(
323       proj.columns.map((col) => (typeof col === 'string') ? null : col)
324     )
325
326     const lhsColumns: Rel.Column[] = []
327     const rhsColumns: Rel.Column[] = []
328     const lhsExtras: Rel.Column[] = []
329     const rhsExtras: Rel.Column[] = []
330
331     for (const col of projColumns) {
332       if (!col) continue
333       if (lhsInv.has(col.target as any))
334         lhsColumns.push(col)
335       if (rhsInv.has(col.target as any))
336         lhsColumns.push(col)
337     }
338     if (joinCondInv)
339       for (const col of joinCondInv) {
340         const inLhs = inArray(col, lhsColumns)
341         const inRhs = inArray(col, rhsColumns)
342         if (!(inLhs || inRhs)) {
343           const newCol
344             = (col instanceof Catalog.Column)
345               ? new Rel.Column(Rel.Relation.fromCata(col.relation), col)
346               : new Rel.Column(null, col)
347           if (lhsInv.has(col))
348             lhsExtras.push(newCol)
349           if (rhsInv.has(col))
350             rhsExtras.push(newCol)
351         }
352       }
353
354     if (!lhsExtras.length && !rhsExtras.length) {
355       // condition only involves attributes in projection list
356       return new Rel.Join(
357         new Rel.Projection(lhsColumns, proj.args.lhs),
358         new Rel.Projection(rhsColumns, proj.args.rhs),
359         joinCond
360       )
```

```
361   } else {
362     arrayExtend(lhsColumns, lhsExtras)
363     arrayExtend(rhsColumns, rhsExtras)
364     return new Rel.Projection(
365       proj.columns,
366       new Rel.Join(
367         new Rel.Projection(lhsColumns, proj.args.lhs),
368         new Rel.Projection(rhsColumns, proj.args.rhs),
369         joinCond
370       )
371     )
372   }
373 }
374 
375 function checkSetCommutativity(op: Rel.Operation) {
376   return ((op.lhs instanceof Rel.HLR) &&
377           (op.rhs instanceof Rel.HLR) &&
378           inArray(op.op, ['union', 'intersect']))
379 }
380 
381 /** Transformation Rule #8: Commutativity of set operations */
382 function commuteSetOperation(op: Rel.PairingOperation, returnNew = false) {
383   const lhs = op.lhs
384   const rhs = op.rhs
385 
386   if (returnNew)
387     return new Rel.Operation(op.op, dupe(rhs), dupe(lhs))
388 
389   return Object.assign(op, {
390     lhs: rhs,
391     rhs: lhs
392   })
393 }
394 
395 function _joinishType(ish: Rel.Joinish | any) {
396   if (ish instanceof Rel.Join) {
397     if (ish.condition instanceof Rel.Conditional)
398       return 'join'
399     else
400       return ish.condition
401   } else if (ish instanceof Rel.Operation &&
402              inArray(ish.op, PairingStrings)) {
403     return ish.op as PairingString
404   }
405   return null
406 }
407 
408 type JoinishAssociativity = false | 'left' | 'right' | 'both'
409 
410 /**
411  * if 'right' then it can be associated 'right', i.e. clockwise; etc
412  */
413 function checkJoinishAssociativity(ish: Rel.Joinish): JoinishAssociativity {
414   const type = _joinishType(ish)
415 
416   if (type === 'join')
417     // TODO: support theta join associativity
418     return false
419   if (!type || !inArray(type, PairingStrings))
```

```
    return false

  let yes = 0
  if (_joinishType(ish.lhs) === type)
    yes += 1
  if (_joinishType(ish.rhs) === type)
    yes += 2

  switch (yes) {
    case 1: return 'right'
    case 2: return 'left'
    case 3: return 'both'
    default: return false
  }
}

/** Transformation Rule #9: Associativity of âŃĹ, ÃŮ, âĹł, and âĹľ
 *  No way to perform destructively; always returns new without dupe.
 */
function associateJoinish<T extends Rel.Joinish>(
    ish: T,
    assoc: 'left' | 'right', // direction to rotate
    returnNew = false) {

  const type = _joinishType(ish)
  // TODO: support theta associativity
  if (type === 'join')
    throw new Error("Association of theta joins not yet supported")
  else if (inArray(type, ['left', 'right', 'except', null]))
    throw new Error("Invalid join type for association")

  const [newInnerLhs, newInnerRhs]
    = (assoc === 'left')
      ? [ish.lhs, (ish.rhs as Rel.Joinish).lhs]
      : [(ish.lhs as Rel.Joinish).rhs, ish.rhs]

  const newInner
    = (ish instanceof Rel.Join)
      ? new Rel.Join(newInnerLhs, newInnerRhs, ish.condition)
      : new Rel.Operation((ish as Rel.Operation).op, newInnerLhs, newInnerRhs)

  const [newLhs, newRhs] =
    (assoc === 'left')
      ? [newInner, (ish.rhs as Rel.Joinish).rhs]
      : [(ish.lhs as Rel.Joinish).lhs, newInner]

  if (returnNew)
    return (ish instanceof Rel.Join)
      ? dupe(new Rel.Join(newLhs, newRhs, ish.condition))
      : dupe(new Rel.Operation((ish as Rel.Operation).op, newLhs, newRhs))

  return Object.assign(ish, {
    lhs: newLhs,
    rhs: newRhs
  })
}

/** Transformation Rule #10: Commuting ÏČ with set operations.
 *  No way to perform destructively; always returns new without dupe.
```

```
 */
function commuteRestrictionSetDown(restr: Rel.Restriction) {
  const setop = restr.args as Rel.PairingOperation
  if (!(setop instanceof Rel.Operation && inArray(setop.op, PairingStrings)))
    throw new Error("Bad commuteRestrictionSetDown() argument")

  const condCopy = dupe(restr.conditions)
  const {lhs, rhs} = setop

  return new Rel.Operation(
    setop.op,
    new Rel.Restriction(restr.conditions, lhs),
    new Rel.Restriction(condCopy, rhs)
  )
}

/** Transformation Rule #11: The ÏĂ operation commutes with âĹł.
 *  No way to perform destructively; always returns new without dupe.
 */
function commuteProjectionUnionDown(proj: Rel.Projection) {
  const union = proj.args as Rel.PairingOperation
  if (!(union instanceof Rel.Operation) || union.op !== 'union')
    throw new Error("Invalid commuteProjectionUnionDown() argument")

  const colsCopy = proj.columns.slice()
  const {lhs, rhs} = union

  return new Rel.Operation(
    'union',
    new Rel.Projection(proj.columns, lhs),
    new Rel.Projection(colsCopy, rhs)
  )
}

/** Transformation Rule #12: Converting a (ÏČ, ÃŮ) sequence into âŃĹ.
 *  No way to perform destructively; always returns new without dupe.
 */
function combineRestrictionCross(restr: Rel.Restriction) {
  const condition = restr.conditions
  const join = restr.args
  if (!(join instanceof Rel.Join) || join.condition !== 'cross')
    throw new Error("Invalid combineRestrictionCross() argument")

  const isJoin = isJoinCondition(condition,
                                 involves(join.lhs),
                                 involves(join.rhs))

  if (!isJoin) return false

  return new Rel.Join(join.lhs, join.rhs, condition)
}

/** Transformation Rule #13: Pushing ÏČ in conjunction with set difference.
 *  No way to perform destructively; always returns new without dupe.
 */
function pushRestrictionDifference(restr: Rel.Restriction, both = false) {
  const condition = restr.conditions
  const setdiff = restr.args as Rel.PairingOperation
  if (!(setdiff instanceof Rel.Operation) || setdiff.op !== 'except')
```

```
538     throw new Error("Invalid pushRestrictionDifference() argument")
539
540   const lhs = new Rel.Restriction(condition, setdiff.lhs)
541   const rhs
542     = (both)
543       ? new Rel.Restriction(dupe(condition), setdiff.rhs)
544       : setdiff.rhs
545
546   return new Rel.Operation('except', lhs, rhs)
547 }
548
549 /** Transformation Rule #14: Pushing ÏČ to only one argument in âĹľ.
550  *  No way to perform destructively; always returns new without dupe.
551  *  'to' should be from checkRestJoinCommutativity()
552  */
553 function pushRestrictionIntersection(restr: Rel.Restriction,
554                                      to: 'lhs' | 'rhs') {
555   const condition = restr.conditions
556   const setinter = restr.args as Rel.PairingOperation
557   if (!(setinter instanceof Rel.Operation) || setinter.op !== 'intersect')
558     throw new Error("Invalid pushRestrictionIntersection() argument")
559
560   const lhs
561     = (to === 'rhs')
562       ? setinter.lhs
563       : new Rel.Restriction(condition, setinter.lhs)
564
565   const rhs
566     = (to === 'lhs')
567       ? setinter.rhs
568       : new Rel.Restriction(condition, setinter.rhs)
569
570   return new Rel.Operation('intersect', lhs, rhs)
571 }
```

2.3.6 src/parser/sqlToRel.ts

```
1
2  import {Rel, Sql, Catalog, OrderingCondition} from './types'
3
4  type RelationLookup = Map<string, Rel.Relation>
5
6  /* bubble a join/relation up to the calling function, also returning
7     the 'realOperation' that took place */
8  class RelationBubbleUp<T> {
9    realOperation: T
10   relationish: Rel.HighLevelRelationish
11
12   constructor(realOp: T, relationish: Rel.HighLevelRelationish) {
13     this.realOperation = realOp
14     this.relationish = relationish
15   }
16 }
17
18 class RenameBubbleUp {
19   target: Rel.Columnish
20   output: string
```

```
21
22    constructor(target: Rel.Columnish, output: string) {
23      if (!(target instanceof Rel.Column))
24        console.log("Rename of non-column!", target, output)
25      this.target = target
26      this.output = output
27    }
28  }
29
30  class ColumnLookup {
31    readonly map: Map<string, Rel.Column[]>
32    readonly catalog: Catalog.Catalog
33    readonly relations: RelationLookup
34
35    constructor(catalog: Catalog.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: Rel.Column): Rel.Column {
42      const cols = this.map.get(name)
43      if (!(target instanceof Rel.Column)) {
44        target = new Rel.Column(null, target, name)
45      }
46      if (!cols)
47        this.map.set(name, [target])
48      else
49        cols.push(target)
50      return target
51    }
52
53    lookup(columnName: string, relationName?: string, as?: string): Rel.Column {
54      if (relationName) {
55        // column references a relation
56        if (!this.relations.has(relationName)) {
57          throw new Error(`Unknown relation "${relationName}"`)
58        }
59        const relation = this.relations.get(relationName) as Rel.Relation
60        const catRelation = this.catalog.relations.get(relation.name) as Catalog.
              Relation
61        // if(!catRelation)
62        //   throw new Error(`${relationName} not in catalog`)
63        if (catRelation.columns.has(columnName))
64          return new Rel.Column(relation,
65                                catRelation.columns.get(columnName) as
                                    Catalog.Column,
66                                as)
67        else
68          throw new Error(`${catRelation.name} doesn't contain ${columnName}`)
69      } else {
70        // implicit relation reference
71        if (this.map.has(columnName)) {
72          // already in the map
73          const cols = this.map.get(columnName) as Rel.Column[]
74          if (cols.length > 1)
75            throw new Error(`Ambiguous column name reference "${columnName}"`)
76
77          return cols[0].alias(as)
```

```
78
79        }
80        // not in map; search for columnName
81        // -console.group()
82        // -console.info(`Searching for ${columnName}`)
83        for (const val of this.relations.values()) {
84          // if (!this.catalog.relations.has(val.name)) {
85          //   throw new Error(`${val.name} not in catalog`)
86          // }
87          const catRel = this.catalog.relations.get(val.name) as Catalog.Relation
88          // -console.info(`${val.name} in catalog, looking for ${columnName}`)
89          if (!catRel.columns.has(columnName))
90            continue
91          // -console.info(`found`)
92          // -console.groupEnd()
93          const col = catRel.columns.get(columnName) as Catalog.Column
94          return new Rel.Column(val, col, as)
95        }
96        // -console.info(`not found`)
97        // -console.groupEnd()
98        throw new Error(`Unknown column ${columnName}`)
99      }
100   }
101 }
102
103 function _joinArgHelper(hs: Sql.Join | Sql.Relation,
104                         relations: RelationLookup,
105                         columns: ColumnLookup,
106                         catalog: Catalog.Catalog,
107                         arg: Sql.Join,
108                         side): Rel.Relation | Rel.Join {
109   if (hs instanceof Sql.Join)
110     return fromJoin(hs, relations, columns, catalog)
111   else if (hs instanceof Sql.Relation)
112     return fromRelation(hs, relations, columns, catalog) as Rel.Relation
113   console.error(`bad join arg ${side}`, arg, "lookup:", relations)
114   throw new Error("Bad join argument lhs")
115 }
116
117 function fromJoin(arg: Sql.Join,
118                   relations: RelationLookup,
119                   columns: ColumnLookup,
120                   catalog: Catalog.Catalog): Rel.Join {
121   const lhs = _joinArgHelper(arg.lhs, relations, columns, catalog, arg, 'left')
122   const rhs = _joinArgHelper(arg.rhs, relations, columns, catalog, arg, 'right'
      )
123   let cond: any = null
124   if (arg.condition) {
125     if (arg.condition instanceof Sql.Conditional)
126       cond = fromConditional(relations, columns, catalog, arg.condition)
127     else if (Array.isArray(arg.condition) && arg.condition.length === 2)
128       cond = fromTargetList(relations, columns, catalog, arg.condition[1])
129     else {
130       console.error("bad conditional", arg, "lookup:", relations)
131       throw new Error("bad conditional")
132     }
133   } else {
134     switch (arg.joinType) {
135       case "join":
```

```
136       case null:
137         cond = "cross"
138         break
139       case "leftouter":
140         cond = "left"
141         break
142       case "rightouter":
143         cond = "right"
144         break
145       case "fullouter":
146         throw new Error("full outer join not supported")
147       // case "natural" | "equi" | null:
148     }
149   }
150
151   const J = new Rel.Join(lhs, rhs, cond)
152   return J
153 }
154
155 function fromColumn(relations: RelationLookup,
156                     columns: ColumnLookup,
157                     catalog: Catalog.Catalog,
158                     arg: Sql.Column,
159                     relHint?: Rel.HighLevelRelationish
160   ): RenameBubbleUp | Rel.Columnish {
161   const alias = arg.alias
162   let target
163   if (arg.target instanceof Sql.Column) {
164     // column of column; either rename it or return target
165     target = fromColumn(relations, columns, catalog, arg.target, relHint)
166     if (!alias)
167       console.log("Why double column?")
168     else if (target instanceof RenameBubbleUp) {
169       console.error("Double rename; arg,target =", arg, target)
170       throw new Error("Double rename not supported")
171     }
172   } else if (typeof(arg.target) === 'string') {
173     // column based on a name
174     target = columns.lookup(arg.target,
175                             arg.relation || undefined,
176                             arg.as || undefined)
177   } else if (arg.target instanceof Sql.Literal) {
178     target = fromLiteral(arg.target)
179   } else if (arg.target instanceof Sql.AggFunction) {
180     target = fromAggFunction(relations, columns, catalog, arg.target, relHint)
181   } else {
182     throw new Error("Unexpected type in column")
183   }
184
185   if (alias) {
186     target = columns.addAlias(alias, target)
187     return new RenameBubbleUp(target, alias)
188   }
189   return target
190 }
191
192 function fromTargetList(relationLookup: RelationLookup,
193                         columnLookup: ColumnLookup,
194                         catalog: Catalog.Catalog,
```

```
195                          targetColumns: Sql.Column[],
196                          relHint?: Rel.HighLevelRelationish
197   ): [Rel.Columnish[], RenameBubbleUp[]] {
198   console.info("fromTargetList:", targetColumns)
199   const renames: RenameBubbleUp[] = []
200   const cols = targetColumns.map((colarg) => {
201     const col = fromColumn(relationLookup,
202                            columnLookup,
203                            catalog,
204                            colarg,
205                            relHint)
206     if (col instanceof RenameBubbleUp) {
207       renames.push(col)
208       return col.target
209     }
210     return col
211   })
212   return [cols, renames]
213 }
214
215 function fromRelation(arg: Sql.Relation,
216                       relations: RelationLookup,
217                       columns: ColumnLookup,
218                       catalog: Catalog.Catalog): Rel.Rename | Rel.Relation |
                             Rel.Join {
219   if (typeof(arg.target) === 'string') {
220     let relat
221     if (relations.has(arg.target))
222       relat = relations.get(arg.target)
223     else if (catalog.relations.has(arg.target)) {
224       relat = new Rel.Relation(arg.target,
225                     catalog.relations.get(arg.target) as Catalog.Relation)
226       relations.set(arg.target, relat)
227     } else {
228       console.error(`Unknown relation ${arg.target}`, arg, relations)
229       throw new Error(`Unknown relation ${arg.target}`)
230     }
231
232     if (arg.alias) {
233       const ren = new Rel.Rename(relat, arg.alias, relat)
234       relations.set(arg.alias, relat)
235       return ren
236     }
237     return relat
238   } else if (arg.target instanceof Sql.Relation) {
239     const relat = fromRelation(arg.target, relations, columns, catalog) as Rel.
          Relation
240     if (!arg.alias)
241       return relat
242     const ren = new Rel.Rename(relat, arg.alias, relat)
243     relations.set(arg.alias, relat)
244     return ren
245   } else if (arg.target instanceof Sql.Join) {
246     const J = fromJoin(arg.target, relations, columns, catalog)
247     if (!arg.alias)
248       return J
249     else
250       throw new Error("Renaming joins not supported ")
251     // const ren = new Rel.Rename()
```

```
252   } else {
253     console.error("bad arg.target type", arg, "lookup:", relations)
254     throw new Error("bad arg.target type")
255   }
256 }
257
258 function fromRelationList(arg: Sql.RelationList,
259                           relations: RelationLookup,
260                           columns: ColumnLookup,
261                           catalog: Catalog.Catalog) {
262   if (arg instanceof Sql.Relation)
263     return fromRelation(arg, relations, columns, catalog)
264   else
265     return fromJoin(arg, relations, columns, catalog)
266 }
267
268 function fromLiteral(lit: Sql.Literal) {
269   switch (lit.literalType) {
270     case 'string':
271       return `'${lit.value}'`
272     case 'number':
273     case 'boolean':
274     case 'null':
275       return String(lit.value)
276     default:
277       throw new Error(`Unknown literal type ${lit.literalType} for ${lit.value}
          `)
278   }
279 }
280
281 function fromAggFunction(rels: RelationLookup,
282                          cols: ColumnLookup,
283                          cata: Catalog.Catalog,
284                          agg: Sql.AggFunction,
285                          relHint?: Rel.HighLevelRelationish) {
286   switch (agg.fname) {
287     case 'count':
288       if (agg.expr === '*' || (agg.expr as Sql.TargetClause).targetlist === '*'
          )
289         return new Rel.RelFunction('count', '*', relHint)
290       else
291         throw new Error("Counting columns not supported")
292     case 'avg':
293     case 'max':
294     case 'min':
295     case 'sum':
296       if (!(agg.expr instanceof Sql.Column))
297         throw new Error(`non-column arguments to aggregates not supported`)
298       const expr = fromColumn(rels, cols, cata, agg.expr, relHint)
299       if (!(expr instanceof Rel.Column)) {
300         console.log("Anomalous AggFunction expr:", expr, "agg:", agg)
301         return new Rel.RelFunction(agg.fname,
302                                    expr as any as Rel.Column,
303                                    relHint)
304       }
305       return new Rel.RelFunction(agg.fname, expr, relHint)
306     default:
307       throw new Error(`Unknown aggregate function ${agg.fname}`)
308   }
```

```
}

function fromOperation(rels: RelationLookup,
                       cols: ColumnLookup,
                       cata: Catalog.Catalog,
                       arg: Sql.Operation,
                       relHint?: Rel.HighLevelRelationish) {
  const lhs = _condArgHelper(rels, cols, cata, arg.lhs, relHint)
  const rhs = _condArgHelper(rels, cols, cata, arg.rhs, relHint)
  return new Rel.Operation(arg.op, lhs, rhs)
}

/* takes an Operand argument */
function _condArgHelper(rels: RelationLookup,
                        cols: ColumnLookup,
                        cata: Catalog.Catalog,
                        hs: Sql.Conditional | Sql.OperandType,
                        relHint?: Rel.HighLevelRelationish) {
  if (hs instanceof Array)
    return fromTargetList(rels, cols, cata, hs, relHint)[0]
  if (hs instanceof Sql.Conditional)
    return fromConditional(rels, cols, cata, hs, relHint)
  else if (hs instanceof Sql.Select)
    return fromSqlSelect(hs, cata)
  // Operand
  else if (hs instanceof Sql.Literal)
    return fromLiteral(hs)
  else if (hs instanceof Sql.AggFunction)
    return fromAggFunction(rels, cols, cata, hs, relHint)
  else if (hs instanceof Sql.Column)
    return fromColumn(rels, cols, cata, hs, relHint)
  else if (hs instanceof Sql.Operation)
    return fromOperation(rels, cols, cata, hs, relHint)
  else
    throw new Error(`Unknown conditional arg type ${hs}`)
}

function _handleSubquery(arg, lhs, op, relations, columns, catalog) {

  const tmpRhs = (arg.rhs instanceof Sql.SelectPair)
                 ? fromSelectPair(arg.rhs, catalog)
                 : fromSqlSelect(arg.rhs, catalog)

  if (op === 'in')
    op = 'eq'

  // lhs = check-against
  // rhs = Selectish
  if (!(tmpRhs instanceof Rel.Projection))
    throw new Error("'in' subqueries must select columns")

  const rhsTarget = tmpRhs.columns

  let conditional: Rel.Conditional
  if (rhsTarget.length > 1)
    conditional = rhsTarget.reduce((L, R) =>
                    new Rel.Conditional(op, L, R), lhs)
  else
    conditional = new Rel.Conditional(op, lhs, rhsTarget[0])
```

```
368
369   return new RelationBubbleUp<Rel.Conditional>(conditional, tmpRhs.args)
370 }
371
372 function fromConditional(relations: RelationLookup,
373                          columns: ColumnLookup,
374                          catalog: Catalog.Catalog,
375                          arg: Sql.Conditional,
376                          relHint?: Rel.HighLevelRelationish
377   ): Rel.Conditional | RelationBubbleUp<Rel.Conditional> {
378   let binOp = true
379   let op: Rel.ThetaOp
380   switch (arg.operation) {
381     case 'not':
382     case 'isnull':
383     case 'exists':
384       binOp = false
385       // break
386     /* binary ops */
387     case 'like':
388     case 'between':
389       throw new Error(`"${arg.operation}" condition not yet supported`)
390
391     case 'or':
392     case 'and':
393     case 'in':
394     case '<':
395     case '>':
396       op = arg.operation
397       break
398     case '<>':
399     case '!=':
400       op = 'neq'
401       break
402     case '<=':
403       op = 'leq'
404       break
405     case '>=':
406       op = 'geq'
407       break
408     case '=':
409       op = 'eq'
410       break
411     default:
412       throw new Error(`Unknown op "${arg.operation}"`)
413   }
414   let lhs = _condArgHelper(relations, columns, catalog, arg.lhs, relHint)
415   if (lhs instanceof RenameBubbleUp) {
416     lhs = lhs.target
417   }
418
419   if (op === 'in' && arg.rhs instanceof Array) {
420     const rs = arg.rhs.map((R) => {
421       const tcond = _condArgHelper(relations, columns, catalog, R, relHint)
422       if (tcond instanceof RenameBubbleUp)
423         return tcond.target
424       return tcond
425     })
426     const cond = new Rel.Conditional('in', lhs, rs)
```

```
427       if (arg.not)
428         throw new Error("'not' conditional is not supported")
429       return cond
430     }
431     if (arg.rhs instanceof Sql.Select ||
432         arg.rhs instanceof Sql.SelectPair) {
433       return _handleSubquery(arg, lhs, op, relations, columns, catalog)
434     }
435     if (op === 'in') {
436       throw new Error("'in' argument should be array or subquery")
437     }
438
439     if (!binOp || !arg.rhs)
440       throw new Error("unary operators not supported")
441     let rhs = _condArgHelper(relations, columns, catalog, arg.rhs, relHint)
442     if (rhs instanceof RenameBubbleUp)
443       rhs = rhs.target
444
445     const condit = new Rel.Conditional(op, lhs, rhs)
446
447     if (arg.not)
448       throw new Error("'not' conditional is not supported")
449     return condit
450   }
451
452   function fromOrdering(rels: RelationLookup,
453                         cols: ColumnLookup,
454                         cata: Catalog.Catalog,
455                         ordering: Sql.Ordering
456     ): Rel.Ordering {
457     const [col, cond] = ordering
458     let column = fromColumn(rels, cols, cata, col)
459     if (column instanceof RenameBubbleUp)
460       column = column.target
461     if (column instanceof Rel.RelFunction)
462       throw new Error("Ordering by function is not supported")
463     return [column, cond]
464   }
465
466   function fromOrderings(orderings: Sql.Ordering[] | null,
467                          rels: RelationLookup,
468                          cols: ColumnLookup,
469                          cata: Catalog.Catalog): Rel.Ordering[] | null {
470     if (!orderings || !orderings.length)
471       return null
472     return orderings.map(fromOrdering.bind(null, rels, cols, cata))
473   }
474
475   export function fromSelectPair(selPair: Sql.SelectPair,
476                                  catalog: Catalog.Catalog) {
477     const lhs = fromSqlSelect(selPair.lhs, catalog)
478     let rhs
479     if (selPair.rhs instanceof Sql.Select)
480       rhs = fromSqlSelect(selPair.rhs, catalog)
481     else
482       rhs = fromSelectPair(selPair.rhs, catalog)
483
484     if (lhs instanceof Rel.Projection &&
485         rhs instanceof Rel.Projection) {
```

```
    if (lhs.columns.length !== rhs.columns.length)
      throw new Error(`Joining on unequal degrees: ` +
                      `${lhs.columns.length} vs ${rhs.columns.length}`)
    const newLhs = lhs.args
    const newRhs = rhs.args
    const newColumns = lhs.columns
    const args = new Rel.Operation(selPair.pairing, newLhs, newRhs)
    return new Rel.Projection(newColumns, args)
  }

  const operation = new Rel.Operation(selPair.pairing, lhs, rhs)
  return operation
}

function _renameReducer(arg: Rel.HighLevelRelationish, ren: RenameBubbleUp) {
  return new Rel.Rename(ren.target, ren.output, arg)
}

function applyRenameBubbleUps(renames: RenameBubbleUp[],
                              args: Rel.HighLevelRelationish) {
    return renames.reduce(_renameReducer, args)
  }

export function fromSqlSelect(select: Sql.Select, catalog: Catalog.Catalog) {

  // map names to the actual instances
  const relations: RelationLookup = new Map()
  const columns = new ColumnLookup(catalog, relations)

  let fromClause: Rel.HighLevelRelationish
      = fromRelationList(select.from, relations, columns, catalog)

  let targetColumns: '*' | Array<string|Rel.Column|Rel.RelFunction>
  let renames: RenameBubbleUp[] = []
  if (select.what.targetlist === '*')
    targetColumns = '*'
  else {
    [targetColumns, renames] = fromTargetList(relations,
                                              columns,
                                              catalog,
                                              select.what.targetlist,
                                              fromClause
                                             )
  }

  let whereClause: Rel.Conditional|null = null
  if (select.where) {
    const tmpCond = fromConditional(relations, columns, catalog, select.where,
                                    fromClause)
    if (tmpCond instanceof RelationBubbleUp) {
      fromClause = new Rel.Join(fromClause, tmpCond.relationish, 'cross')
      whereClause = tmpCond.realOperation as Rel.Conditional
    } else {
      whereClause = tmpCond
    }
  }

  // let groupBy: Array<string|Rel.Column|Rel.RelFunction>|null = null
  if (select.groupBy) {
```

```
545
546     const gList = fromTargetList(relations, columns, catalog, select.groupBy)
547     if (gList[1].length)
548       console.warn("Ignored rename(s) of GROUP BY clause")
549     const groupBy = gList[0] as Rel.Column[]
550
551     const groupByNames: string[] = []
552     groupBy.forEach((g) => {
553       if (!(g instanceof Rel.Column))
554         throw new Error("Cannot group-by non-column")
555
556       let foundIdx = -1
557       for (let i = 0; i < renames.length; i++) {
558         const ren = renames[i]
559         if (ren.target instanceof Rel.Column && ren.target.target === g.target)
              {
560           foundIdx = i
561           groupByNames.push(ren.output)
562           renames.splice(foundIdx, 1)
563           break
564         }
565       }
566       if (foundIdx < 0) {
567         if (g.as)
568           groupByNames.push(g.as)
569         else {
570           console.error("Bad column:", g)
571           throw new Error("Un 'as'd Column")
572         }
573       }
574     })
575
576     if (targetColumns === '*')
577       throw new Error("Group-By on '*' selection unsupported")
578
579     const functs: Rel.RelFunction[] = []
580     const aggRenames = groupByNames.slice()
581     targetColumns.forEach((col, colIdx) => {
582       if (typeof col === 'string')
583         throw new Error("Group-By on literals unsupported")
584
585       if (col instanceof Rel.Column && col.target instanceof Rel.RelFunction) {
586         const target = col.target
587         const colname = col.as
588         console.log(`col:`, col, renames)
589         if (colname) {
590           aggRenames[colIdx] = colname
591           for (let i = 0; i < renames.length; i++) {
592             const ren = renames[i]
593             if (ren.target instanceof Rel.Column
594                 && ren.target.target instanceof Rel.RelFunction) {
595               if (ren.target.target === target && ren.output === colname) {
596                 console.log("Found it!", i)
597                 renames.splice(i, 1)
598                 break
599               }
600             }
601           }
602         }
```

```
603         console.log(renames)
604         col = col.target
605       }
606
607       if (col instanceof Rel.RelFunction) {
608         functs.push(col)
609         const gFunctName = col.getName()
610         let foundIdx = -1
611         for (let i = 0; i < renames.length; i++) {
612           const ren = renames[i]
613           if (ren.target instanceof Rel.RelFunction
614               && ren.target.getName() === gFunctName) {
615             foundIdx = i
616             aggRenames[colIdx] = ren.output
617             renames.splice(foundIdx, 1)
618             break
619           }
620         }
621         if (foundIdx < 0 && !aggRenames[colIdx])
622           aggRenames[colIdx] = gFunctName
623
624       } else if (col instanceof Rel.Column) {
625         if (!col.as)
626           throw new Error("Un 'as'd column")
627         if (groupByNames.indexOf(col.as) === -1) {
628           throw new Error(`GroupBy confusion; didn't find "${col.as}"`)
629         }
630       } else {
631         console.error("targetColumns:", targetColumns)
632         throw new Error("Unexpected argument in targetColumns")
633       }
634     })
635
636     fromClause = new Rel.Aggregation(groupBy, functs, fromClause, aggRenames)
637
638     // TODO: implement HAVING with aggregate-condition support
639     if (select.having) {
640
641       const havingCond = fromConditional(relations, columns, catalog,
642                                          select.having, fromClause)
643       if (!(havingCond instanceof Rel.Conditional))
644         throw new Error("Unexpected type from fromConditional; RelationBubbleUp
              ?")
645
646       fromClause = new Rel.Restriction(havingCond, fromClause)
647     }
648
649   } else if (select.having)
650     console.warn("Ignored HAVING clause without GROUP BY clause")
651
652   const orderBy = fromOrderings(select.orderBy, relations, columns, catalog)
653
654   if (renames.length) {
655     fromClause = applyRenameBubbleUps(renames, fromClause)
656   }
657
658   const Restrict = whereClause
659       ? new Rel.Restriction(whereClause, fromClause)
660       : fromClause
```

```
661
662   const Project = targetColumns === '*'
663       ? Restrict
664       : new Rel.Projection(targetColumns as Array<string|Rel.Column>, Restrict)
665
666   return Project
667 }
```

2.3.7 src/parser/tests.ts

```
1
2  import {Rel} from './types'
3
4  export const selectTests = [
5
6  `-- Query a
7  SELECT      S.sid, S.sname, S.rating, S.age
8  FROM        Sailors AS S
9  WHERE       S.rating > 7`,
10
11 `-- Query b (invalid)
12 SELECT      S.sid, S.sname
13 FROM        Sailors AS S
14 WHERE       S.color = âĂŸgreenâĂŹ`,
15
16 `--Query c
17 SELECT      B.color
18 FROM        Sailors AS S,  Reserves AS R, Boats AS B
19 WHERE       S.sid=R.sid AND R.bid=B.bid AND S.sname = âĂŸLubberâĂŹ`,
20
21 `--Query d
22 SELECT      sname
23 FROM        Sailors, Boats, Reserves
24 WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND Boats.color=
       âĂŸredâĂŹ
25 UNION
26 SELECT      sname
27 FROM        Sailors, Boats, Reserves
28 WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND Boats.color=
       âĂŸgreenâĂŹ
29 `,
30
31 `--Query e (invalid)
32 -- day is a reserved word
33 SELECT    S.sname
34 FROM       Sailors AS S,  Reserves AS R
35 WHERE    R.sid = S.sid AND R.bid = 100 AND R.rating > 5 AND R.day = âĂŸ8/9/09
       âĂŹ
36 `,
37
38 `--Query f
39 SELECT      sname
40 FROM        Sailors, Boats, Reserves
41 WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
42 Boats.color=âĂŹredâĂŹ
43 INTERSECT
44 SELECT      sname
```

```
45 FROM         Sailors, Boats, Reserves
46 WHERE      Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
47 Boats.color=âĂŹgreenâĂŹ
48 `,
49
50 `--Query g
51 SELECT       S.sid
52 FROM         Sailors AS S,  Reserves AS R, Boats AS B
53 WHERE      S.sid=R.sid AND R.bid=B.bid AND B.color=âĂŸredâĂŹ
54 EXCEPT
55 SELECT       S2.sid
56 FROM         Sailors AS S2,  Reserves AS R2, Boats AS B2
57 WHERE      S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color=âĂŸgreenâĂŹ
58 `,
59
60 `--Query h
61 SELECT      S.sname
62 FROM       Sailors AS S
63 WHERE      S.sid IN ( SELECT   R.sid
64                       FROM     Reserves AS R
65                       WHERE   R.bid = 103)
66 `,
67
68 `--Query i
69 SELECT       S.sname
70 FROM         Sailors AS S
71 WHERE      S.sid IN ((SELECT   R.sid
72           FROM     Reserves AS R, Boats AS B
73           WHERE   R.bid = B.bid AND B.color = âĂŸredâĂŹ)
74           INTERSECT
75           (SELECT   R2.sid
76           FROM     Reserve AS R2, Boats AS B2
77           WHERE   R2.bid = B2.bid AND B2.color = âĂŸgreenâĂŹ))
78 `,
79
80 `--Query j
81 SELECT      S.sname
82 FROM         Sailors S
83 WHERE      EXISTS (SELECT   B.bid
84      FROM     Boats B
85      WHERE   EXISTS (SELECT   R.bid
86        FROM      Reserves R
87        WHERE   R.bid = B.bid AND
88              R.sid = S.sid))
89 `,
90
91 `--Query k
92 SELECT    S.sname
93 FROM        Sailors S
94 WHERE    S.age > (SELECT    MAX (S2.age)
95           FROM       Sailors S2
96           WHERE    S2.rating = 10)
97 `,
98
99 `--Query l
100 SELECT     B.bid, Count (*) AS reservationcount
101 FROM         Boats B, Reserves R
102 WHERE      R.bid=B.bid AND B.color = âĂŸredâĂŹ
103 GROUP BY B.bid
```

```
`,

`--Query m
SELECT      B.bid, Count (*) AS reservationcount
FROM         Boats B, Reserves R
WHERE      R.bid=B.bid AND B.color = âĂŸredâĂŹ
GROUP BY B.bid
HAVING    B.color = âĂŸredâĂŹ
`,

`--Query n (invalid)
SELECT    Sname
FROM       Sailors
WHERE    Sailor.sid   IN (SELECT   Reserves.bid, Reserves.sid
   FROM     Reserves
   CONTAINS
    (SELECT Boats.bid
     FROM  Boats
     WHERE    bname   =  âĂŸinterlakeâĂŹ) )
`,

`--Query o (fixed)
SELECT      S.rating, Avg (S.age) As average
FROM        Sailors S
WHERE     S.age > 18
GROUP BY S.rating
HAVING    Count (*) > 1
`
]

export type ResultTuple = [null|Error|object,
                           null|Error|Rel.HighLevelRelationish]
export const selectResults: ResultTuple[] =
  selectTests.map(() => [null, null] as ResultTuple)
```

2.4 src/parser/peg

2.4.1 src/parser/peg/sql.pegjs

```
/*
  Initially inspired by grammar of the "Phoenix" SQL layer
    (https://forcedotcom.github.io/phoenix/index.html)

  Primarily based on PostgreSql syntax:
    https://www.postgresql.org/docs/9/static/sql-syntax.html
    https://www.postgresql.org/docs/9/static/sql-select.html
    https://github.com/postgres/postgres/blob/master/src/backend/parser/gram.y
*/

start
  = Statements

Statements
  = _ lhs:Statement rhs:( _ ";" _ Statement )* _ ";"?
  { return rhs.reduce((result, element) => result.concat(element[3]), [lhs]) }

Statement
```

```
19    = Selectish
20
21  Selectish
22    = SelectPair
23    / Select
24
25
26  SelectPair
27    = lhs:Select __
28      pairing:$( "UNION"i / "INTERSECT"i / "EXCEPT"i ) __
29      spec:( "ALL"i __ / "DISTINCT"i __ )?
30      rhs:( Selectish )
31    {
32      return new Sql.SelectPair(pairing.toLowerCase(),
33                                spec && spec[0].toLowerCase(),
34                                lhs,
35                                rhs)
36    }
37
38  Select
39    = "SELECT"i __ what:TargetClause __
40      "FROM"i   __ from:FromClause
41      where:(   __ "WHERE"i  __           WhereClause )?
42      groupBy:( __ "GROUP"i  __ "BY"i __ GroupByClause )?
43      having:(  __ "HAVING"i __           HavingClause )?
44      orderBy:( __ "ORDER"i  __ "BY"i __ OrderByClause )?
45    {
46      return new Sql.Select(what, from, where && where[3],groupBy && groupBy[5],
47                            having && having[3], orderBy && orderBy[5])
48    }
49    / "(" _ sel:Select _ ")" { return sel }
50
51  TargetClause
52    = spec:$( "DISTINCT"i __ / "ALL"i __ )?
53      target:(
54        "*"
55        / TargetList
56      )
57    { return {
58        'type': Sql.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:$( "<" / ">" ) { return op }
86      )?
87
88  RelationList
89    = item1:RelationItem _ "," _ items:RelationList
90      { return new Sql.Join(item1, items) }
91      / Join
92      / RelationItem
93
94  RelationItem "RelationItem"
95    = item:RelationThing __ ( "AS"i __ )? alias:Name
96    { return new Sql.Relation(item, alias) }
97    / RelationThing
98
99  RelationThing
100   = "(" _ list:RelationList _ ")"
101   { return list }
102   / "(" _ join:Join _ ")"
103   { return join }
104   / tableName:Name
105   { return new Sql.Relation(tableName) }
106
107 Join
108   = item1:RelationItem __
109     jtype:JoinType __
110     item2:RelationItem
111     jcond:(
112       __ "ON"i
113       __ expr:Condition
114       { return expr }
115       / __ "USING"i _
116         "(" _ list:TargetList _ ")"
117         { return ['using', list] }
118     )?
119   { return new Sql.Join(item1, item2, jtype, jcond) }
120
121 TargetList
122   = item1:TargetItem _ "," _ items:TargetList
123     { return [item1].concat(items) }
124     / item:TargetItem
125     { return [item] }
126
127 TargetItem "TargetItem"
128   = table:Name ".*"
129   { return new Sql.Column(table, '*', `${table}.*`, null) }
130   / op:Operand __ "AS"i __ alias:Name
131   { return new Sql.Column(null, op, alias, alias )}
132   / op:Operand __ alias:Name
133   { return new Sql.Column(null, op, alias, alias )}
134   / op:Operand _ "=" _ alias:Name
135   { return new Sql.Column(null, op, alias, alias) }
136   / op:Operand
```

```
137   { return (op instanceof Sql.Column) ? op : new Sql.Column(null, op) }
138
139 Condition "Condition"
140   = lhs:AndCondition rhs:( __ "OR"i __ Condition )?
141   { return rhs ? new Sql.Conditional('or', lhs, rhs[3]) : lhs }
142
143 AndCondition
144   = lhs:InnerCondition rhs:( __ "AND"i __ AndCondition )?
145   { return rhs ? new Sql.Conditional('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 Sql.Conditional('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 Sql.Conditional('contains', lhs, rhs) }
174
175 ConditionComp "Conditional-Comparison"
176   = lhs:Operand _ cmp:Compare _ rhs:Operand
177   { return new Sql.Conditional(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 Sql.Conditional('in', lhs_op, rhs_ops, not) }
187
188 ConditionExists
189   = "EXISTS"i _
190     "(" _ subquery:Selectish _ ")"
191   { return new Sql.Conditional('exists', subquery) }
192
193 ConditionLike
194   = lhs_op:Operand __
195     not:( "NOT"i __ )?
```

```
196       "LIKE"i __
197       rhs_op:Operand
198     { return new Sql.Conditional('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 Sql.Conditional('between', lhs_op, rhs, not) }
219
220   ConditionNull
221     = lhs:Operand __ "IS"i __
222       not:( "NOT"i __ )?
223       NullLiteral
224     { return new Sql.Conditional('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 Sql.Column(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 Sql.AggFunction("avg", term) }
250
251   AggFunctionCount
252     = "COUNT"i _
253       "(" _
254         targ:TargetClause _
```

```
255       ")"
256     { return new Sql.AggFunction("count", targ) }
257
258   AggFunctionMax
259     = "MAX"i _
260       "(" _
261         term:Term _
262       ")"
263     { return new Sql.AggFunction("max", term) }
264
265   AggFunctionMin
266     = "MIN"i _
267       "(" _
268         term:Term _
269       ")"
270     { return new Sql.AggFunction("min", term) }
271
272   AggFunctionSum
273     = "SUM"i _
274       "(" _
275         term:Term _
276       ")"
277     { return new Sql.AggFunction("sum", term) }
278
279   /***** PRIMITIVES *****/
280
281   Name
282     = DQStringLiteral
283       / BTStringLiteral
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 Sql.Operation("||",
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 Sql.Operation(rh[1],
310                                                                  lh, rh[3])) }
311
312   Factor // literal | function | Operand | column | makeOperation
313     = lhs:Term
```

```
    rhs:( _ ("*" / "/") _ Term ) *
  { return reduceIfRHS(lhs, rhs, (lh, rh) => new Sql.Operation(rh[1],
                                                               lh, rh[3])) }

Compare
  = "<>"
    / "<="
    / ">="
    / "="
    / "<"
    / ">"
    / "!="

JoinType "JoinType"
  = ( "CROSS"i __ )? "JOIN"i
  { return "join" }
  / "INNER"i __ "JOIN"i
  { return "equi" }
  / "NATURAL"i __ "JOIN"i
  { return "natural" }
  / "LEFT"i __ ( "OUTER"i __ )? "JOIN"i
  { return "left" }
  / "RIGHT"i __ ( "OUTER"i __ )? "JOIN"i
  { return "right" }
  / "FULL"i __ ( "OUTER"i __ )? "JOIN"i
  { return "full" }

/***** LITERALS *****/

Literal "Literal"
  = SQStringLiteral
    / NumericLiteral
    / ExponentialLiteral
    / BooleanLiteral
    / NullLiteral

BTStringLiteral "backtick string"
  = $( '`' ( [^`] / '``' )+ '`' )

DQStringLiteral "double-quote string"
  = $( '"' ( [^"] / '""' )+ '"' )

SQStringLiteral "single-quote string"
  = lit:$( "'" ( [^'] / "''" )* "'" !SQStringLiteral )
  { return new Sql.Literal('string', lit.slice(1, -1)) }
  / lit:$( ("âĂŸ"/"âĂŹ") ( [^âĂŹ] )* "âĂŹ" ) // fancy single-quote
  { return new Sql.Literal('string', lit.slice(1, -1)) }

ExponentialLiteral "exponential"
  = val:$( NumericLiteral "e" IntegerLiteral )
  { return new Sql.Literal('number', parseFloat(val)) }

NumericLiteral "number"
  = IntegerLiteral
    / DecimalLiteral

IntegerLiteral "integer"
  = int:$( "-"? [0-9]+ )
  { return new Sql.Literal('number', parseInt(int)) }
```

```
373
374 DecimalLiteral "decimal"
375   = value:$( "-"? [0-9]+ "." [0-9]+ )
376   { return new Sql.Literal('number', parseFloat(value)) }
377
378 NullLiteral "null"
379   = "NULL"i
380   { return new Sql.Literal('null', null) }
381
382 BooleanLiteral "boolean"
383   = TruePrim
384     / FalsePrim
385
386 TruePrim
387   = "TRUE"i
388   { return new Sql.Literal('boolean', true) }
389
390 FalsePrim
391   = "FALSE"i
392   { return new Sql.Literal('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   = $("ABS"i / "ALL"i / "ALLOCATE"i / "ALTER"i / "AND"i / "ANY"i / "ARE"i /
415         "ARRAY_AGG"i / "ARRAY"i / "ASENSITIVE"i / "ASYMMETRIC"i / "AS"i /
416         "ATOMIC"i / "AT"i / "AUTHORIZATION"i / "AVG"i
417       / "BEGIN"i / "BETWEEN"i / "BIGINT"i / "BINARY"i / "BLOB"i / "BOOLEAN"i /
418         "BOTH"i / "BY"i
419       / "CALLED"i / "CALL"i / "CARDINALITY"i / "CASCADED"i / "CASE"i / "CAST"i
            /
420         "CEILING"i / "CEIL"i / "CHARACTER_LENGTH"i / "CHAR_LENGTH"i /
421         "CHARACTER"i / "CHAR"i / "CHECK"i / "CLOB"i / "CLOSE"i / "COALESCE"i /
422         "COLLATE"i / "COLLECT"i / "COLUMN"i / "COMMIT"i / "CONDITION"i /
423         "CONNECT"i / "CONSTRAINT"i / "CONVERT"i / "CORRESPONDING"i / "CORR"i /
424         "COUNT"i / "COVAR_POP"i / "COVAR_SAMP"i / "CREATE"i / "CROSS"i /
425         "CUBE"i / "CUME_DIST"i / "CURRENT_CATALOG"i / "CURRENT_DATE"i /
426         "CURRENT_DEFAULT_TRANSFORM_GROUP"i / "CURRENT_PATH"i / "CURRENT_ROLE"i
              /
427         "CURRENT_SCHEMA"i / "CURRENT_TIMESTAMP"i / "CURRENT_TIME"i /
428         "CURRENT_TRANSFORM_GROUP_FOR_TYPE"i / "CURRENT_USER"i / "CURRENT"i /
429         "CURSOR"i / "CYCLE"i
```

```
430       / "DATALINK"i / "DATE"i / "DAY"i / "DEALLOCATE"i / "DECIMAL"i /
431         "DECLARE"i / "DEC"i / "DEFAULT"i / "DELETE"i / "DENSE_RANK"i /
432         "DEREF"i / "DESCRIBE"i / "DETERMINISTIC"i / "DISCONNECT"i /
433         "DISTINCT"i / "DLNEWCOPY"i / "DLPREVIOUSCOPY"i / "DLURLCOMPLETE"i /
434         "DLURLCOMPLETEONLY"i / "DLURLCOMPLETEWRITE"i / "DLURLPATHONLY"i /
435         "DLURLPATHWRITE"i / "DLURLPATH"i / "DLURLSCHEME"i / "DLURLSERVER"i /
436         "DLVALUE"i / "DOUBLE"i / "DROP"i / "DYNAMIC"i
437       / "EACH"i / "ELEMENT"i / "ELSE"i / "END-EXEC"i / "END"i / "ESCAPE"i /
438         "EVERY"i / "EXCEPT"i / "EXECUTE"i / "EXEC"i / "EXISTS"i / "EXP"i /
439         "EXTERNAL"i / "EXTRACT"i
440       / "FALSE"i / "FETCH"i / "FILTER"i / "FIRST_VALUE"i / "FLOAT"i / "FLOOR"i
              /
441         "FOREIGN"i / "FOR"i / "FREE"i / "FROM"i / "FULL"i / "FUNCTION"i /
442         "FUSION"i
443       / "GET"i / "GLOBAL"i / "GRANT"i / "GROUPING"i / "GROUP"i
444       / "HAVING"i / "HOLD"i / "HOUR"i
445       / "IDENTITY"i / "IMPORT"i / "INDICATOR"i / "INNER"i / "INOUT"i /
446         "INSENSITIVE"i / "INSERT"i / "INTEGER"i / "INTERSECTION"i /
447         "INTERSECT"i / "INTERVAL"i / "INTO"i / "INT"i / "IN"i / "IS"i
448       / "JOIN"i
449       / "LAG"i / "LANGUAGE"i / "LARGE"i / "LAST_VALUE"i / "LATERAL"i /
450         "LEADING"i / "LEAD"i / "LEFT"i / "LIKE_REGEX"i / "LIKE"i / "LN"i /
451         "LOCALTIMESTAMP"i / "LOCAL"i / "LOCALTIME"i / "LOWER"i
452       / "MATCH"i / "MAX_CARDINALITY"i / "MAX"i / "MEMBER"i / "MERGE"i /
453         "METHOD"i / "MINUTE"i / "MIN"i / "MODIFIES"i / "MODULE"i / "MOD"i /
454         "MONTH"i / "MULTISET"i
455       / "NATIONAL"i / "NATURAL"i / "NCHAR"i / "NCLOB"i / "NEW"i / "NONE"i /
456         "NORMALIZE"i / "NOT"i / "NO"i / "NTH_VALUE"i / "NTILE"i / "NULLIF"i /
457         "NULL"i / "NUMERIC"i
458       / "OCCURRENCES_REGEX"i / "OCTET_LENGTH"i / "OFFSET"i / "OF"i / "OLD"i /
459         "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         "REFERENCING"i / "REF"i / "REGR_AVGX"i / "REGR_AVGY"i / "REGR_COUNT"i /
466         "REGR_INTERCEPT"i / "REGR_R2"i / "REGR_SLOPE"i / "REGR_SXX"i /
467         "REGR_SXY"i / "REGR_SYY"i / "RELEASE"i / "RESULT"i / "RETURNS"i /
468         "RETURN"i / "REVOKE"i / "RIGHT"i / "ROLLBACK"i / "ROLLUP"i / "ROWS"i /
469         "ROW_NUMBER"i / "ROW"i
470       / "SAVEPOINT"i / "SCOPE"i / "SCROLL"i / "SEARCH"i / "SECOND"i /
471         "SELECT"i / "SENSITIVE"i / "SESSION_USER"i / "SET"i / "SIMILAR"i /
472         "SMALLINT"i / "SOME"i / "SPECIFICTYPE"i / "SPECIFIC"i /
473         "SQLEXCEPTION"i / "SQLSTATE"i / "SQLWARNING"i / "SQL"i / "SQRT"i /
474         "START"i / "STATIC"i / "STDDEV_POP"i / "STDDEV_SAMP"i / "SUBMULTISET"i
              /
475         "SUBSTRING_REGEX"i / "SUBSTRING"i / "SUM"i / "SYMMETRIC"i /
476         "SYSTEM_USER"i / "SYSTEM"i
477       / "TABLESAMPLE"i / "TABLE"i / "THEN"i / "TIMESTAMP"i / "TIMEZONE_HOUR"i /
478         "TIMEZONE_MINUTE"i / "TIME"i / "TO"i / "TRAILING"i /
479         "TRANSLATE_REGEX"i / "TRANSLATE"i / "TRANSLATION"i / "TREAT"i /
480         "TRIGGER"i / "TRIM_ARRAY"i / "TRIM"i / "TRUE"i / "TRUNCATE"i
481       / "UESCAPE"i / "UNION"i / "UNIQUE"i / "UNKNOWN"i / "UNNEST"i / "UPDATE"i
              /
482         "UPPER"i / "USER"i / "USING"i
483       / "VALUES"i / "VALUE"i / "VARBINARY"i / "VARCHAR"i / "VARYING"i /
484         "VAR_POP"i / "VAR_SAMP"i
```

```
      / "WHENEVER"i / "WHEN"i / "WHERE"i / "WIDTH_BUCKET"i / "WINDOW"i /
        "WITHIN"i / "WITHOUT"i / "WITH"i
      / "XMLAGG"i / "XMLATTRIBUTES"i / "XMLBINARY"i / "XMLCAST"i /
        "XMLCOMMENT"i / "XMLCONCAT"i / "XMLDOCUMENT"i / "XMLELEMENT"i /
        "XMLEXISTS"i / "XMLFOREST"i / "XMLITERATE"i / "XMLNAMESPACES"i /
        "XMLPARSE"i / "XMLPI"i / "XMLQUERY"i / "XMLSERIALIZE"i / "XMLTABLE"i /
        "XMLTEXT"i / "XMLVALIDATE"i / "XML"i
      / "YEAR"i
  ) !Ident
```

2.4.2 src/parser/peg/relations.pegjs

```

start
  = _ rel:Relations _
  { return rel }

Relations
  = lhs:Relation
    rhs:( _ Relations )*
  { return rhs.reduce((l, r) => l.concat(r[1]), [lhs]) }

Relation
  = table:Name
    _ "(" _
      cols:Columns
    _ ")"
  { return [table, cols] }

Columns
  = lhs:Column rhs:( _ "," _ Column )*
  { return rhs.reduce((l,r) => l.concat([r[3]]), [lhs]) }

Column
  = name:Name _ ":" _ typ:Ident
  { return [name, typ] }


/* sql primitives */

Name "Name"
  = DQStringLiteral
    / BTStringLiteral
    / Ident

Ident "UnquotedIdent"
  = $( [A-Za-z_][A-Za-z0-9_]* )

BTStringLiteral "backtick string"
  = $( '`' ( [^`] / '``' )+ '`' )

DQStringLiteral "double-quote string"
  = $( '"' ( [^"] / '""' )+ '"' )

_ "OptWhitespace"
  = WS* Comment? WS* {}

```

```
__ "ReqWhitespace"
  = WS+ Comment? WS* {}
    / WS* Comment? WS+ {}

WS
  = [ \t\n]

Comment "Comment"
  = "/*" ( !"*/" . )* "*/"   {}
    / "--" ( !"\n" . )* "\n" {}
```

2.5 src/parser/types

2.5.1 src/parser/types/index.ts

```
1
2  import * as Rel from './Rel'
3  import * as Sql from './Sql'
4  import * as Catalog from './Catalog'
5
6  export {Rel, Sql, Catalog}
7
8  export type JoinString = "join"       // "," | "JOIN" | "CROSS JOIN"
9                         | "equi"       // "INNER JOIN" | "JOIN ... USING"
10                        | "natural"    // "NATURAL JOIN"
11                        | "leftouter"  // "LEFT [OUTER] JOIN"
12                        | "rightouter" // "RIGHT [OUTER] JOIN"
13                        | "fullouter"  // "FULL [OUTER] JOIN"
14
15 export type OrderingCondition = "asc" | "desc" | "<" | ">"
16
17 export type PairingString = 'union' | 'intersect' | 'except'
18 export type PairingCondition = 'all' | 'distinct' | null
19
20 export type OperationOps = '||' | '+' | '-' | '*' | '/'
21
22 export type AggFuncName = 'avg' | 'count' | 'max' | 'min' | 'sum'
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 }
```

2.5.2 src/parser/types/Catalog.ts

```
1
2  type NameTypePair = [string, string]
3  type RelnameColsPair = [string, NameTypePair[]]
4  type ColumnMap = Map<string, Column>
5
```

```
export class Catalog {

  static fromParse(relations: RelnameColsPair[]) {
    const rels = new Map()
    relations.forEach((ele) => {
      const [tname, cols] = ele
      const columnMap = new Map() as ColumnMap
      const newRelation = new Relation(tname, columnMap)
      cols.forEach(Column.fromNameTypePair.bind(null, columnMap, newRelation))
      rels.set(tname, newRelation)
    })
    return new Catalog(rels)
  }

  readonly relations: Map<string, Relation>

  constructor(relations: Map<string, Relation>) {
    this.relations = relations
  }
}

export class Relation {
  readonly name: string
  readonly columns: ColumnMap

  constructor(name: string, columns: ColumnMap) {
    this.name = name
    this.columns = columns
  }
}

export class Column {

  static fromNameTypePair(columnMap: ColumnMap,
                          newRelation: Relation,
                          col: NameTypePair): Column {
    const newCol = new Column(col[0], col[1], newRelation)
    columnMap.set(col[0], newCol)
    return newCol
  }

  readonly name: string
  readonly typ: string
  readonly relation: Relation

  constructor(name: string, typ: string, relation: Relation) {
    this.name = name
    this.typ = typ
    this.relation = relation
  }
}
```

2.5.3 src/parser/types/Rel.ts

```
import {OperationOps, PairingString, AggFuncName, OrderingCondition
        } from './index'
import * as Catalog from './Catalog'
```

```
4
5  // literals are strings
6  export type OperandType = string | Operation | Column
7  export type Ordering = [Column | string, OrderingCondition]
8  export type ColumnValueType = Catalog.Column | RelFunction | string
9  export type Columnish = Column | RelFunction | string
10 export type Joinish = Join | PairingOperation
11 export type HighLevelRelationish = Relation | Join | Restriction | Projection |
12                                   Rename | Operation | Aggregation
13
14 /* enumerated strings attached to class instances.
15  * Useful for JSON output.
16  * CAN be used for quick comparison, but **instanceof should be prefered**.
17  */
18 export const enum TypeString {
19   Aggregation = "aggregation",
20   Restriction = "restriction",
21   Projection = "projection",
22   Rename = "rename",
23   Relation = "relation",
24   Column = "column",
25   Conditional = "conditional",
26   Join = "join",
27   Function = "funct",
28   Operation = "op"
29 }
30
31 export const enum HLRTypeString {
32   Aggregation = TypeString.Aggregation,
33   Restriction = TypeString.Restriction,
34   Projection = TypeString.Projection,
35   Rename = TypeString.Rename,
36   Relation = TypeString.Relation,
37   Join = TypeString.Join,
38   Operation = TypeString.Operation
39 }
40
41 export abstract class HLR {
42   readonly abstract type: HLRTypeString
43
44   constructor() {}
45 }
46
47 export class Operation extends HLR {
48   readonly type = HLRTypeString.Operation
49   op: OperationOps | PairingString
50   lhs: OperandType | HighLevelRelationish
51   rhs: OperandType | HighLevelRelationish
52
53   constructor(op: OperationOps | PairingString,
54               lhs: OperandType | HighLevelRelationish,
55               rhs: OperandType | HighLevelRelationish) {
56     super()
57     this.op = op
58     this.lhs = lhs
59     this.rhs = rhs
60   }
61 }
62
```

```
63  export interface PairingOperation extends Operation {
64    op: PairingString
65    lhs: HighLevelRelationish
66    rhs: HighLevelRelationish
67  }
68
69  export class Column {
70    readonly type = TypeString.Column
71    relation: Relation | null
72    target: ColumnValueType
73    as: string | null
74
75    constructor(relation: Relation | null,
76                target: ColumnValueType,
77                As: string | null = null) {
78      this.relation = relation
79      this.target = target
80      this.as = As || null
81    }
82
83    alias(alias?: string) {
84      if (!alias)
85        return this
86      return new Column(this.relation, this.target, alias)
87    }
88  }
89
90  export class RelFunction {
91    readonly type = TypeString.Function
92    fname: AggFuncName
93    expr: '*' | Column // TODO: support correct args
94    hlr?: HighLevelRelationish
95
96    constructor(fname: AggFuncName,
97                expr: '*' | Column,
98                hlr?: HighLevelRelationish) {
99      this.fname = fname
100     this.expr = expr
101     if (hlr)
102       this.hlr = hlr
103   }
104
105   getName(): string {
106     if (this.expr === '*')
107       return `${this.fname}_*`
108     else if (this.expr.as)
109       return `${this.fname}_${this.expr.as}`
110     else {
111       console.error("Failure to name RelFunction", this)
112       throw new Error("Couldn't produce name for RelFunction")
113     }
114   }
115 }
116
117 export class Aggregation extends HLR {
118   readonly type = HLRTypeString.Aggregation
119   attributes: Column[]
120   functions: RelFunction[]
121   relation: HighLevelRelationish
```

```
122   renames: string[]
123
124   constructor(attributes: Column[], functions: RelFunction[],
125               relation: HighLevelRelationish, renames: string[]) {
126     super()
127     this.attributes = attributes
128     this.functions = functions
129     this.relation = relation
130     this.renames = renames
131
132     if (renames.length &&
133         renames.length !== (attributes.length + functions.length)) {
134       const [rl, al, fl] = [renames.length, attributes.length, functions.length
            ]
135       console.error("Bad number of renames;", renames, attributes, functions)
136       throw new Error(`Bad number of renames; ${rl} != ${al} + ${fl}`)
137     }
138   }
139 }
140
141 export type ThetaOp = 'eq' | 'neq' | 'leq' | 'geq' | '<' | '>' | 'and' | 'or' |
142                       'in'
143 export type ConditionalArgumentType = OperandType | RelFunction | Conditional
144
145 export class Conditional {
146   readonly type = TypeString.Conditional
147   operation: ThetaOp
148   lhs: ConditionalArgumentType
149   rhs: ConditionalArgumentType | OperandType[]
150
151   constructor(op: ThetaOp,
152               lhs: ConditionalArgumentType,
153               rhs: ConditionalArgumentType | OperandType[]) {
154     this.operation = op.toLowerCase() as ThetaOp
155     this.lhs = lhs
156     this.rhs = rhs
157   }
158 }
159
160 export class Restriction extends HLR {
161   readonly type = HLRTypeString.Restriction
162   conditions: Conditional
163   args: HighLevelRelationish
164
165   constructor(conditions: Conditional, args: HighLevelRelationish) {
166     super()
167     this.conditions = conditions
168     this.args = args
169   }
170 }
171
172 export class Projection extends HLR {
173   readonly type = HLRTypeString.Projection
174   columns: Array<string|Column>
175   args: HighLevelRelationish
176
177   constructor(columns: Array<string|Column>, args: HighLevelRelationish) {
178     super()
179     this.columns = columns
```

```
180       this.args = args
181     }
182   }
183
184   type _RenameInputType = Relation | Column | RelFunction |
185                           Rename | string
186
187   export class Rename extends HLR {
188     readonly type = HLRTypeString.Rename
189     input: _RenameInputType
190     output: string
191     args: HighLevelRelationish
192
193     constructor(input: _RenameInputType,
194                 output: string,
195                 args: HighLevelRelationish) {
196       super()
197       this.input = input
198       this.output = output
199       this.args = args
200     }
201   }
202
203   export class Relation extends HLR {
204     static fromCata(target: Catalog.Relation): Relation {
205       return new Relation(target.name, target)
206     }
207
208     readonly type = HLRTypeString.Relation
209     readonly name: string
210     readonly target: Catalog.Relation
211
212     constructor(name: string, target: Catalog.Relation) {
213       super()
214       this.name = name
215       this.target = target
216     }
217   }
218
219   export type JoinCond = "cross" | "left" | "right" | Conditional
220
221   // cross
222   // natural (no condition)
223   // theta join (with condition)
224   // semi (left and right)
225   export class Join extends HLR {
226     readonly type = HLRTypeString.Join
227     lhs: HighLevelRelationish
228     rhs: HighLevelRelationish
229     condition: JoinCond
230
231     constructor(lhs: HighLevelRelationish,
232                 rhs: HighLevelRelationish,
233                 cond: JoinCond) {
234       super()
235       this.lhs = lhs
236       this.rhs = rhs
237       this.condition = cond
238     }
```

```
}
```

2.5.4 src/parser/types/Sql.ts

```
import {OrderingCondition, OperationOps, PairingString, PairingCondition,
        JoinString, AggFuncName} from './index'

export const LITERAL_TYPE       = "literal"
export const COLUMN_TYPE        = "column"
export const JOIN_TYPE          = "join"
export const RELATION_TYPE      = "relation"
export const CONDITIONAL_TYPE   = "conditional"
export const AGGFUNCTION_TYPE   = "aggfunction"
export const OPERATION_TYPE     = "operation"
export const SELECTCLAUSE_TYPE  = "selectclause"
export const TARGETCLAUSE_TYPE  = "targetclause"
export const SELECTPAIR_TYPE    = "selectpair"

export type Ordering = [Column, OrderingCondition]

export type RelationList = Relation | Join
type TargetList = Column[]

export interface TargetClause {
  type: "targetclause"
  spec: PairingCondition
  targetlist: "*" | TargetList
}

export class Literal {
  readonly type = LITERAL_TYPE
  literalType: 'string' | 'number' | 'boolean' | 'null'
  value: string | number | boolean | null

  constructor(literalType: 'string' | 'number' | 'boolean' | 'null',
              value: string | number | boolean | null) {
    this.literalType = literalType
    this.value = value
  }
}

export type Selectish = Select | SelectPair

export class SelectPair {
  readonly type = SELECTPAIR_TYPE
  pairing: PairingString
  condition: PairingCondition
  lhs: Select
  rhs: Selectish

  constructor(pairing: PairingString,
              condition: PairingCondition,
              lhs: Select,
              rhs: Selectish) {
    this.pairing = pairing
    this.condition = condition || null
    this.lhs = lhs
```

```
54      this.rhs = rhs
55    }
56  }
57
58  export class Select {
59    readonly SELECTCLAUSE_TYPE
60    what: TargetClause
61    from: RelationList
62    where: Conditional | null
63    groupBy: TargetList | null
64    having: Conditional | null
65    orderBy: Ordering[] | null
66
67    constructor(what: TargetClause,
68                from: RelationList,
69                where: Conditional | null,
70                groupBy: TargetList | null,
71                having: Conditional | null,
72                orderBy: Ordering[] | null) {
73      this.what = what
74      this.from = from
75      this.where = where
76      this.groupBy = groupBy
77      this.having = having
78      this.orderBy = orderBy
79    }
80  }
81
82  export type OperandType = Literal | AggFunction | Column |
83                        Operation | string
84
85  export class Column {
86    readonly type = COLUMN_TYPE
87    relation: string | null
88    target: OperandType
89    as: string | null
90    alias: string | null
91
92    constructor(relation: string | null,
93                target: OperandType,
94                As: string | null = null,
95                alias: string | null = null) {
96      this.relation = relation
97      this.target = target
98      this.as = As || null
99      this.alias = alias || null
100   }
101 }
102
103 export class Join {
104   readonly type = JOIN_TYPE
105   joinType: JoinString
106   condition: Conditional | ['using', TargetList] | null
107   lhs: Join | Relation
108   rhs: Join | Relation
109
110   constructor(lhs: Join | Relation,
111               rhs: Join | Relation,
112               joinType: JoinString = 'join',
```

```
113               condition: Conditional | ['using', TargetList] | null = null
114   ) {
115     this.lhs = lhs
116     this.rhs = rhs
117     this.joinType = joinType || 'join'
118     this.condition = condition || null
119   }
120 }
121
122 export class Relation {
123   readonly type = RELATION_TYPE
124   target: Relation | Join | string
125   alias: string | null
126
127   constructor(target: Relation | Join | string,
128               alias: string | null = null) {
129     this.target = target
130     this.alias = alias || null
131   }
132 }
133
134 export type ConditionalOp = 'or' | 'and' | 'not' | 'in' | 'exists' | 'like' |
135                             'between' | 'isnull' | '<>' | 'contains' |
136                             '<=' | '>=' | '=' | '<' | '>' | '!='
137
138 export class Conditional {
139   readonly type = CONDITIONAL_TYPE
140   operation: ConditionalOp
141   lhs: Conditional | OperandType
142   rhs: Conditional | OperandType | null
143   not: boolean
144
145   constructor(operation: ConditionalOp,
146               lhs: Conditional | OperandType,
147               rhs: Conditional | OperandType | null = null,
148               not: boolean = false) {
149     if (operation === 'in' && lhs instanceof Array && lhs.length === 1)
150       lhs = lhs[0]
151     this.operation = operation
152     this.lhs = lhs
153     this.rhs = rhs || null
154     this.not = not
155   }
156 }
157
158 export class AggFunction {
159   readonly type = AGGFUNCTION_TYPE
160   fname: AggFuncName
161   expr: OperandType | TargetClause
162
163   constructor(fname: AggFuncName, expr: OperandType | TargetClause) {
164     this.fname = fname
165     this.expr = expr
166   }
167 }
168
169 export class Operation {
170   readonly type = OPERATION_TYPE
171   op: OperationOps
```

```
  lhs: OperandType
  rhs: OperandType

  constructor(op: OperationOps, lhs: OperandType, rhs: OperandType) {
    this.op = op
    this.lhs = lhs
    this.rhs = rhs
  }
}
```

2.6 src/query_tree

2.6.1 src/query_tree/node.ts

```
import {QTOperation, Relation, Join, Restriction, Projection, Rename,
        Operation, Aggregation} from './operation'

import {Rel} from '../parser/types'

// if RelRelation:    just name
// if RelJoin:        ....
// if RelRestriction: SYM _ (conditions)
// if RelProjection:  SYM _ (columns)
// if RelRename:      SYM _ (A / B)
// if RelOperation:   hlr SYM hlr

export default class Node {
  hlr: Rel.HighLevelRelationish
  operation: QTOperation
  children: Node[] = []
  depth: number = 0

  constructor(hlr: Rel.HighLevelRelationish, depth: number = 0) {
    this.hlr = hlr
    this.depth = depth
    this.generateOpAndKids()
  }

  generateOpAndKids() {
    if (this.hlr instanceof Rel.Relation) {
      this.operation = new Relation(this.hlr)
    } else if (this.hlr instanceof Rel.Join) {
      this.operation = new Join(this.hlr)
      this.addNode(new Node(this.hlr.lhs, this.depth + 1))
      this.addNode(new Node(this.hlr.rhs, this.depth + 1))
    } else if (this.hlr instanceof Rel.Restriction) {
      this.operation = new Restriction(this.hlr)
      this.addNode(new Node(this.hlr.args, this.depth + 1))
    } else if (this.hlr instanceof Rel.Projection) {
      this.operation = new Projection(this.hlr)
      this.addNode(new Node(this.hlr.args, this.depth + 1))
    } else if (this.hlr instanceof Rel.Rename) {
      this.operation = new Rename(this.hlr)
      this.addNode(new Node(this.hlr.args, this.depth + 1))
    } else if (this.hlr instanceof Rel.Aggregation) {
      this.operation = new Aggregation(this.hlr)
      this.addNode(new Node(this.hlr.relation, this.depth + 1))
```

```
    } else if (this.hlr instanceof Rel.Operation) {
      this.operation = new Operation(this.hlr)
      this.addNode(new Node(this.hlr.lhs as Rel.HighLevelRelationish, this.
          depth + 1))
      this.addNode(new Node(this.hlr.rhs as Rel.HighLevelRelationish, this.
          depth + 1))
    } else {
      console.error("Unknown type", this.hlr)
      throw new Error("Unknown op type")
    }
  }

  addNode(node: Node) {
    node.depth = this.depth + 1
    this.children.push(node)
  }
}
```

2.6.2 src/query_tree/operation.tsx

```
import * as React from 'react'
import {Rel} from '../parser/types'
import {htmlRelRelation, htmlRelProjection, relJoinHelper, htmlRelRestriction,
        htmlRelRename, getSymbol, htmlRelAggregation
       } from '../parser/relationalText'

// if RelRelation:     just name
// if RelJoin:         ....
// if RelRestriction: SYM _ (conditions)
// if RelProjection:  SYM _ (columns)
// if RelRename:       SYM _ (A / B)
// if RelOperation:   hlr SYM hlr

export class QTOperation {
  symbolName: string
  hlr: Rel.HighLevelRelationish
  html: JSX.Element

  constructor(hlr: Rel.HighLevelRelationish) {
    this.hlr = hlr
  }
}

export class Relation extends QTOperation {
  hlr: Rel.Relation
  constructor(hlr: Rel.Relation) {
    super(hlr)
    this.html = htmlRelRelation(hlr)
  }
}

export class Join extends QTOperation {
  hlr: Rel.Join
  constructor(hlr: Rel.Join) {
    super(hlr)
    this.html = this.generateHTML()
  }
```

```
38
39    generateHTML() {
40      const [joinSymbol, cond] = relJoinHelper(this.hlr)
41      return (
42        <span className="RelJoin">
43          <span className="operator">{joinSymbol}</span>
44          {cond}
45        </span>
46      )
47    }
48  }
49
50  export class Restriction extends QTOperation {
51    hlr: Rel.Restriction
52    constructor(hlr: Rel.Restriction) {
53      super(hlr)
54      this.html = htmlRelRestriction(hlr, true)
55    }
56  }
57
58  export class Projection extends QTOperation {
59    hlr: Rel.Projection
60    constructor(hlr: Rel.Projection) {
61      super(hlr)
62      this.html = htmlRelProjection(hlr, true)
63    }
64  }
65
66  export class Rename extends QTOperation {
67    hlr: Rel.Rename
68    constructor(hlr: Rel.Rename) {
69      super(hlr)
70      this.html = htmlRelRename(hlr, true)
71    }
72  }
73
74  export class Operation extends QTOperation {
75    hlr: Rel.Operation
76    constructor(hlr: Rel.Operation) {
77      super(hlr)
78      this.html = this.generateHTML()
79    }
80
81    generateHTML() {
82      const SYM = getSymbol(this.hlr.op)
83      return (
84        <span className="operator">{SYM}</span>
85      )
86    }
87  }
88
89  export class Aggregation extends QTOperation {
90    hlr: Rel.Aggregation
91    constructor(hlr: Rel.Aggregation) {
92      super(hlr)
93      this.html = htmlRelAggregation(hlr, true)
94    }
95  }
96
```

```
/*export class From extends Operation {
  constructor() {
    super("From")
  }

  addTarget(data) {
     if(data.lhs && data.rhs) {
       this.addTarget(data.lhs)
       this.addTarget(data.rhs)
       return
     }

     else if(data.lhs || data.rhs) {
       throw new Error('From without both lhs and rhs')
     }

    let arg = data.target.target
    if(data.alias) arg += ` as ${data.alias}`
    this.addArgument(arg)
  }
}*/

/*export class Where extends Operation {
  constructor() {
    super("Where")
  }

  addTarget(data) {
    let lhs = this.getArgument(data.lhs)
    let rhs = this.getArgument(data.rhs)
    this.addArgument(lhs + ` ${data.operation} ` + rhs)
  }

  getArgument(data): string {
    if(data.lhs && data.rhs) {
      let lhs = this.getArgument(data.lhs)
      let rhs = this.getArgument(data.rhs)
      let arg = lhs + ` ${data.operation} ` + rhs
      return arg
    } else if(data.lhs || data.rhs) {
      throw new Error('lhs and rhs not both specified')
    }

    let arg
    if(data.relation) arg = `${data.relation}.${data.target}`
    if(data.relation && data.alias) arg += ` as ${data.alias}`
    if(data.literalType === "number" && data.value) arg = data.value
    if(data.literalType === "string" && data.value) arg = `\'${data.value}\'`

    return arg
  }
}*/
```