CS 5300 Project #1

Jared Rainwater & Samuel K. Grush

October 31, 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 `Statements`. A `Statement` can be either a `Select` or `SelectPair`. `Select` is broken up into 6 clauses: `TargetClause`, `FromClause`, `WhereClause`, `GroupByClause`, `HavingClause` and `OrderByClause`. These correspond to all the possibilities of a valid SQL Select statement. A `SelectPair` is two 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 `TargetItems`. A `TargetItem` is a column-like specifier; it can be a relation name with ".*" or an `Operand` with optional alias.

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

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

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

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

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

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

The `ReservedWord` rule contains 340 keywords that the ISO/ANSI SQL:2008 standard states are **never allowed as identifiers**. This set is almost certainly overkill, as most SQL implementations only reserve a *small* fraction of it. It is also excessively large, making up over $^1/_3$ of the grammar's sourcecode and **90%** of the uncompressed compiled grammar.

1.2 Interpretation

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

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

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

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

2 Source Code

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

2.1 src/

2.1.1 src/index.ts

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

import './styles/tests.scss'

import Main from './Main'

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 | null

  debug: string
}

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

  constructor(props: any) {
    super(props)
    this.state = {
      queryInputText: "",
      status: "",
      catalog: null,
      queryJSON: null,
      relJSON: null,
      debug: ""
    }

    this.onRelationsInputUpdate = this.onRelationsInputUpdate.bind(this)
    this.onQueryInputUpdate = this.onQueryInputUpdate.bind(this)

  }

  onRelationsInputUpdate(output: RelationsInputOutput) {
    if (output.error) {
      this.setState({
        catalog: null,
        status: `Error Parsing Relations:  ${output.error}`,
        debug: output.traceback
      })
    } else {
      this.setState({
        catalog: output.catalog,
        status: "Successfully Parsed Relations",
        debug: ''
      })
    }
  }

  onQueryInputUpdate(text: string): void {
    this.setState({
      status: "Parsing Query...",
      queryInputText: text,
      queryJSON: null,
      relJSON: null,
      debug: ""
    })

  }

  /*parseQuery(): void {
    const {queryInputText, catalog} = this.state

    let queryJSON
    let relJSON

    const tracer = new Tracer(queryInputText, {
      useColor: false,
      showTrace: true
    })
    try {
      queryJSON = parseSql(queryInputText, {tracer})
      let root = parseSQLToTree(queryJSON)
      this.setState({
        queryJSON,
        root,
        status: "Query parsed; Generating relational algebra..."
      })
    } catch (ex) {
      const err: SqlSyntaxError = ex
      this.setState({
        status: `Error Parsing Query:  ${err.message}`,
        debug: tracer.getParseTreeString()
      })
      throw err
    }

    try {
      relJSON = sqlToRelationalAlgebra(queryJSON, catalog as Catalog)
      this.setState({
        relJSON,
        status: "Generated relational algebra"
      })
    } catch (ex) {
      this.setState({
        status: `Error Generating Relational Algebra: ${ex.message}`,
      })
      throw ex
    }
  }*/

  render() {
    return (
      <main id="main">
        <RelationsInput onUpdate={this.onRelationsInputUpdate} />
        <QueryInput
          onUpdate={this.onQueryInputUpdate}
          disabled={!this.state.catalog}
        />
        <div id="parse-status">{this.state.status}</div>
        <div id="main-output">
          <TestCase
            catalog={this.state.catalog}
            queryInputText={this.state.queryInputText}
            doRun={true} // bad idea??
            anchor="main-test"
            name="Main Test"
          />
          <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

```
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 | null
  error: null | Error
  traceback: '' | string
}

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

```

```
interface RelationsInputState {
  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|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 {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 parseSQLToTree from '../query_tree/parse'
import Tree from '../components/tree'

interface TestCaseProps {
  catalog: Catalog | null
  queryInputText: string
  doRun: boolean
  anchor: string
  name?: string
}

interface TestCaseState {
  status: string
  treeStatus: string
  queryJSON: any
  relAlJSON: any
  root: Node | null
  relAlHTML: JSX.Element | null
  color: string
  tscolor: string
  debug: any
}

export default class TestCase extends React.Component<TestCaseProps,
    TestCaseState> {
  constructor(props) {
    super(props)
    this.state = this.initialState()
    this.run = this.run.bind(this)
  }

  componentDidMount() {
    this.propsReceived(this.props)
  }

  componentWillReceiveProps(newProps: TestCaseProps) {
    this.propsReceived(newProps)
  }

  propsReceived(newProps: TestCaseProps) {
    const {catalog, queryInputText, doRun} = this.props
    if (newProps.catalog !== catalog ||
        newProps.queryInputText !== queryInputText ||
        newProps.doRun !== doRun
       ) {
      this.setState(this.initialState(), () => {
        if (newProps.catalog && newProps.queryInputText && newProps.doRun)
          this.run(newProps)
      })
    }
  }

  initialState(): TestCaseState {
    return {
      status: 'init',
      treeStatus: '',
      queryJSON: null,
      relAlJSON: null,
      relAlHTML: null,
      root: null,
      color: 'currentcolor',
      tscolor: 'currentcolor',
      debug: ''
    }
  }

  run(props: TestCaseProps = this.props) {

    const catalog = props.catalog as Catalog

    const tracer = new Tracer(props.queryInputText, {
      useColor: false,
      showTrace: true
    })

    let status = ''
    let treeStatus = ''
    let queryJSON = null
    let relAlJSON = null
    let relAlHTML = null
    let root: Node | null = null
    let color = 'currentcolor'
    let tscolor = 'currentcolor'
    let debug = ''

    try {
      queryJSON = parseSql(props.queryInputText, {tracer})
      status = "SQL Scanned and Tokenized"
      color = "green"
    } catch (ex) {
      if (ex instanceof SqlSyntaxError)
        status = `Parser Syntax Error: ${ex.message}`
      else
        status = `Other Parser ${ex}`
      console.error(ex)
      color = "red"
      debug = tracer.getParseTreeString()
    }

    if (queryJSON) {
      try {
        relAlJSON = sqlToRelationalAlgebra(queryJSON, catalog) as any
        status = "SQL Parsed and converted to Relational Algebra"
        color = "green"
      } catch (ex) {
        status = `Relational Algebra ${ex}`
        color = "red"
        console.error(ex)
      }
    }
    if (relAlJSON) {
      try {
        relAlHTML = htmlHLR(relAlJSON)
        status = "Relational Algebra rendered to HTML"
        color = "green"
      } catch (ex) {
        status = `HTML Conversion Error: ${ex}`
        color = "red"
        console.error(ex)
      }
      try {
        root = parseSQLToTree(/*relAlJSON*/queryJSON)
        treeStatus = "Tree Generated"
        tscolor = "green"
      } catch (ex) {
        treeStatus = `Tree Error: ${ex}`
        tscolor = "red"
        console.error(ex)
      }
    }

    this.setState({
      status,
      treeStatus,
      queryJSON,
      relAlJSON,
      relAlHTML,
      root,
      color,
      tscolor,
      debug
     })
  }

  render() {
    return (
      <section id={this.props.anchor} className="testcase">
        <hr />
        <h3>{this.props.name || this.props.anchor}</h3>
        <pre><code>{this.props.queryInputText}</code></pre>
        <div className="testcase-status">
          <span style={{color: this.state.color}}>
            Status: {this.state.status || "OK"}
          </span>
          { this.state.treeStatus && (
            <span style={{color: this.state.tscolor}}>
              Tree Status: {this.state.treeStatus}
            </span>
          )}
        </div>
        <div className="testcase-inner">
          <div className="relal-html" data-empty={!this.state.relAlHTML}>
            <h4>Relational Algebra</h4>
            {this.state.relAlHTML}
          </div>
          <div className="sql-json" data-empty={!this.state.queryJSON}>
            <h4>SQL Structure</h4>
            <JSONPretty json={this.state.queryJSON} />
          </div>
          <div className="relal-json" data-empty={!this.state.relAlJSON}>
            <h4>Relational Algebra Structure</h4>
            <JSONPretty json={this.state.relAlJSON} />
          </div>
          <div className="tree" data-empty={!this.state.root}>
            <h4>Tree</h4>
            { this.state.root &&
                <Tree root={this.state.root} margin={10} />
            }
          </div>
          <div className="traceback" data-empty={!this.state.debug}>
            <h4>Error Traceback</h4>
            <pre><code>{this.state.debug}</code></pre>
          </div>
        </div>
      </section>
    )
  }
}
```

2.2.4 src/components/Tests.tsx

```
import * as React from "react"

import {Catalog} from '../parser/types'
import TestCase from './TestCase'
import {selectTests} from "../parser/tests"

export function getTestName(testStr: string) {
  if (testStr.startsWith('--'))
    return testStr.split("\n", 1)[0].slice(2).trim()
  return ''
}

interface TestsProps {
  catalog: Catalog | null
}

interface TestsState {
  catalog: Catalog | null
  doRun: boolean
  queryNames: string[]
}

export default class Tests extends React.Component<TestsProps, TestsState> {
  constructor(props) {
    super(props)
    this.state = {
      catalog: props.Catalog,
      doRun: false,
      queryNames: selectTests.map(getTestName)
    }

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

  componentWillReceiveProps(nextProps: TestsProps) {
    const catalog = nextProps.catalog
    if (catalog !== this.props.catalog)
      this.setState({
        catalog,
        doRun: false
      })
  }

  run(e?) {
    if (e) e.preventDefault()
    if (this.state.catalog)
      this.setState({
        doRun: true
      })
  }

  render() {
    return (
      <div id="tests-div">
        <h2>Test Cases</h2>
        <button
          onClick={this.run}
          disabled={!this.state.catalog}
        >Run Tests</button>
        <nav id="tests-nav">
          <ol>
            {
              this.state.queryNames.map((qName, idx) => {
                const anchor = `#q${idx}`
                return (
                  <li key={anchor}>
                    <a href={anchor}>{qName || anchor}</a>
                  </li>
                )
              })
            }
          </ol>
        </nav>
        <div id="tests-list">
          {
            selectTests.map((testStr, idx) => (
              <TestCase
                queryInputText={testStr}
                catalog={this.state.catalog}
                doRun={this.state.doRun}
                key={idx}
                anchor={`q${idx}`}
                name={this.state.queryNames[idx] || undefined}
              />
            ))
          }
        </div>
      </div>
    )
  }
}
```

2.2.5 src/components/tree.tsx

```
import * as React from 'react'
import Node from 'query_tree/node'
import '../styles/tree.scss'

interface TreeProps {
  root: Node
  margin: number
}

interface TreeState {
}

export default
class Tree extends React.Component<TreeProps, TreeState> {
  render() {
      const rows: JSX.Element[] = []
      let node = this.props.root
      let depth = 0
      while (node) {
        const row = <TreeRow
                      node={node}
                      key={depth}
                      offset={this.props.margin * depth}/>
        rows.push(row)
        depth++
        node = node.children[0]
      }
      return (
      <div>
        {rows}
      </div>
      )
  }
}

interface TreeRowProps {
  offset: number
  node: Node
}

interface TreeRowState {
}

class TreeRow extends React.Component<TreeRowProps, TreeRowState> {
  render() {
    return (
      <div className="tree-row" style={{paddingLeft: this.props.offset}}>
```

```
        {this.props.node.operation.name} ({this.props.node.operation.arguments.
            map(arg => {
          return arg + ", "
        })})
      </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 * as types from './types'
import {fromSqlSelect, fromSelectPair} from './sqlToRel'

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

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

export function sqlToRelationalAlgebra(sqlStatements, catalog: types.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 types.SqlSelect)
    return fromSqlSelect(TLStatement, catalog)
  else if (TLStatement instanceof types.SqlSelectPair)
    return fromSelectPair(TLStatement, catalog)
  else
    throw new Error(`Unknown sqlToRelationalAlgebra arg ${TLStatement}`)
}
```

2.3.2 src/parser/relationalText.tsx

```
import * as React from 'react'

import * as types from './types'

export function getSymbol(input: string) {
  switch (input) {
    // passthroughs
    case '||':
    case '+':
    case '-':
```

```
    case '*':
    case '/':
    case '<':
    case '>':
      return input

    case 'restriction':
      return "ÏČ"
    case 'projection':
      return "Îă"
    case 'rename':
      return "ÏĄ"
    case 'rename-divider':
      return "âĹŢ"

    case 'union':
      return "âĹł"
    case 'intersect':
      return "âĹľ"
    case 'except':
      return "âĹŠ"

    case 'join':
      return "âŃĹ"
    case 'left':
    case 'ljoin':
      return "âŃĽ"
    case 'right':
    case 'rjoin':
      return "âŃŁ"
    case 'cross':
    case 'crossjoin':
      return "âĺĽ"
    case 'divide':
      return "Ãů"

    case 'eq':
      return "="
    case 'neq':
      return "âĽă"
    case 'leq':
      return "âĽď"
    case 'geq':
      return "âĽě"
    case 'and':
      return "âĹğ"
    case 'or':
      return "âĹĺ"
    case 'in':
      return "âĹŁ"
    default:
      throw new Error(`Unknown symbol name "${input}"`)
  }
}

export function htmlRelRestriction(res: types.RelRestriction) {
  const SYM = getSymbol('restriction')
  const COND = htmlRelConditional(res.conditions)
  const ARGS = htmlHLR(res.args)
  return (
    <span className="RelRestriction">
      <span className="operator">{SYM}</span>
      <sub className="condition">
        {COND}
      </sub>
      (
        <span className="HLR">
          {ARGS}
        </span>
      )
    </span>
  )
}

export function htmlRelProjection(res: types.RelProjection) {
  const SYM = getSymbol('projection')
  const COLUMNS: Array<string|HTMLSpanElement> = []
  res.columns.forEach((col, idx) => {
    if (idx > 0)
      COLUMNS.push(",")
    if (col instanceof types.RelColumn)
      COLUMNS.push(htmlRelColumn(col, idx))
    else if ((col as any) instanceof types.RelFunction)
      COLUMNS.push(htmlRelFunction(col, idx))
    else
      COLUMNS.push(col)
  })
  const ARGS = htmlHLR(res.args)
  return (
    <span className="RelProjection">
      <span className="operator">{SYM}</span>
      <sub className="columns">
        {COLUMNS}
      </sub>
      (
        <span className="HLR">
          {ARGS}
        </span>
      )
    </span>
  )
}

export function htmlRelColumn(col: types.RelColumn, iter?: number) {

  if (col.as) {
    return (
      <span className="RelColumn" key={iter}>
        <span className="column-as">{col.as}</span>
      </span>
    )
  }

  if (!col.relation) {
    return (
      <span className="RelColumn" key={iter}>
        <span className="column-name">{getName(col.target)}</span>
      </span>
    )
  }

  return (
    <span className="RelColumn" key={iter}>
      <span className="relation-name">{getName(col.relation)}</span>
      .
      <span className="column-name">{getName(col.target)}</span>
    </span>
  )
}

export function htmlRelFunction(funct: types.RelFunction, idx?) {
  const NAME = funct.fname.toUpperCase()
  const EXPR = funct.expr === '*'
          ? '*'
          : htmlRelColumn(funct.expr)

  return (
    <span className="RelFunction" key={idx}>
      <span className="function-name">{NAME}</span>
      (
        {EXPR}
      )
    </span>
  )
}

export function getName(thing) {
  if (typeof(thing) === 'string')
    return thing
  if (thing instanceof types.RelRelation)
    return thing.name
  if (thing instanceof types.RelColumn)
    return thing.as || htmlRelColumn(thing)
  if (thing instanceof types.RelFunction)
    return htmlRelFunction(thing as types.RelFunction)
  if (thing instanceof types.Column)
    return thing.name
  console.info("getName", thing)
  throw new Error("unexpected thing to getName")
}

export function htmlRelRename(ren: types.RelRename) {
  const SYM = getSymbol('rename')
  const INPUT = getName(ren.input)
  const OUTPUT = ren.output
  const ARGS = htmlHLR(ren.args as types.HighLevelRelationish)

  return (
    <span className="RelRename">
      <span className="operator">{SYM}</span>
      <sub className="condition">
        {OUTPUT} {getSymbol('rename-divider')} {INPUT}
      </sub>
      (
        <span className="HLR">
          {ARGS}
        </span>
      )
    </span>
  )
}

export function htmlRelRelation(rel: types.RelRelation) {
  const NAME = rel.name
  return (
    <span className="RelRelation">
      {NAME}
    </span>
  )
}

export function htmlRelJoin(join: types.RelJoin) {
  let joinSymbol
  let cond
  if (typeof(join.condition) === 'string') {
    joinSymbol = getSymbol(join.condition)
    cond = null
  } else if (join.condition instanceof types.RelConditional) {
    joinSymbol = getSymbol('join')
    cond = htmlRelConditional(join.condition)
  } else {
    throw new Error(`unknown RelJoin condition ${join.condition}`)
  }
  const LHS = htmlHLR(join.lhs)
  const RHS = htmlHLR(join.rhs)

  return (
    <span className="RelJoin">
      {LHS}
      <span className="operator">{joinSymbol}</span>
      {
        cond && (
          <sub className="condition">
            {cond}
          </sub>
        )
      }
      {RHS}
    </span>
  )
}

export function htmlRelOperation(op: types.RelOperation) {
  const OPSYM = getSymbol(op.op)
  const LHS = htmlRelOperand(op.lhs as any)
  const RHS = htmlRelOperand(op.rhs as any)

  return (
    <span className="RelOperation">
      {LHS}
      <span className="operator">{OPSYM}</span>
      {RHS}
    </span>
  )
}

```

```
export function htmlRelOperand(operand: types.RelOperandType) {
  if (typeof(operand) === 'string')
    return operand
  if (operand instanceof types.RelFunction)
    return htmlRelFunction(operand)
  if (operand instanceof types.RelOperation)
    return htmlRelOperation(operand)
  if (operand instanceof types.RelColumn)
    return htmlRelColumn(operand)
  // throw new Error("Unexpected operand type")
  return htmlHLR(operand)
}

export function htmlRelConditional(cond: types.RelConditional) {
  const OPSYM = getSymbol(cond.operation)
  const LHS = cond.lhs instanceof types.RelConditional
          ? htmlRelConditional(cond.lhs)
          : htmlRelOperand(cond.lhs)
  const RHS = cond.rhs instanceof types.RelConditional
          ? htmlRelConditional(cond.rhs)
          : ( cond.rhs instanceof Array
              ? cond.rhs.map(htmlRelOperand)
              : htmlRelOperand(cond.rhs)
            )

  return (
    <span className="RelConditional">
      <span className="lhs">
        {LHS}
      </span>
      <span className="operator">{OPSYM}</span>
      <span className="rhs">
        {RHS}
      </span>
    </span>
  )
}

export function htmlHLR(hlr: types.HighLevelRelationish) {
  if (hlr instanceof types.RelRestriction)
    return htmlRelRestriction(hlr)
  if (hlr instanceof types.RelProjection)
    return htmlRelProjection(hlr)
  if (hlr instanceof types.RelRename)
    return htmlRelRename(hlr)
  if (hlr instanceof types.RelOperation)
    return htmlRelOperation(hlr)
  if (hlr instanceof types.RelRelation)
    return htmlRelRelation(hlr)
  if (hlr instanceof types.RelJoin)
    return htmlRelJoin(hlr)
  console.error("unknown HLR:", hlr)
  throw new Error("Unknown type passed to htmlHLR")
}
```

2.3.3 src/parser/sqlToRel.ts

```

import * as types from './types'

type ColumnValueType = types.RelColumn | types.RelFunction | string

type RelationLookup = Map<string, types.RelRelation>

/* bubble a join/relation up to the calling function, also returning
   the 'realOperation' that took place */
class BubbleUp<T> {
  realOperation: T
  relationish: types.HighLevelRelationish

  constructor(realOp: T, relationish: types.HighLevelRelationish) {
    this.realOperation = realOp
    this.relationish = relationish
  }
}

class RenameBubbleUp {
  target: ColumnValueType
  output: string

  constructor(target: ColumnValueType, output: string) {
    this.target = target
    this.output = output
  }
}

class ColumnLookup {
  readonly map: Map<string, types.RelColumn[]>
  readonly catalog: types.Catalog
  readonly relations: RelationLookup

  constructor(catalog: types.Catalog, relations: RelationLookup, init?) {
    this.map = new Map(init)
    this.catalog = catalog
    this.relations = relations
  }

  addAlias(name: string, target: types.RelColumn) {
    const cols = this.map.get(name)
    if (!cols)
      this.map.set(name, [target])
    else
      cols.push(target)
  }

  lookup(columnName: string, relationName?: string, as?: string): types.
      RelColumn {
    if (relationName) {
      // column references a relation
      if (!this.relations.has(relationName)) {
        throw new Error(`Unknown relation "${relationName}"`)
      }
      const relation = this.relations.get(relationName) as types.RelRelation
      const catRelation = this.catalog.relations.get(relation.name) as types.
          Relation
      // if(!catRelation)
      //   throw new Error(`${relationName} not in catalog`)
      if (catRelation.columns.has(columnName))
        return new types.RelColumn(relation,
                                   catRelation.columns.get(columnName) as types
                                       .Column,
                                   as)
      else
        throw new Error(`${catRelation.name} doesn't contain ${columnName}`)
    } else {
      // implicit relation reference
      if (this.map.has(columnName)) {
        // already in the map
        const cols = this.map.get(columnName) as types.RelColumn[]
        if (cols.length > 1)
          throw new Error(`Ambiguous column name reference "${columnName}"`)

        return cols[0].alias(as)

      }
      // not in map; search for columnName
      console.group()
      console.info(`Searching for ${columnName}`)
      for (const val of this.relations.values()) {
        // if (!this.catalog.relations.has(val.name)) {
        //   throw new Error(`${val.name} not in catalog`)
        // }
        const catRel = this.catalog.relations.get(val.name) as types.Relation
        console.info(`${val.name} in catalog, looking for ${columnName}`)
        if (!catRel.columns.has(columnName))
          continue
        console.info(`found`)
        console.groupEnd()
        const col = catRel.columns.get(columnName) as types.Column
        return new types.RelColumn(val, col, as)
      }
      console.info(`not found`)
      console.groupEnd()
      throw new Error(`Unknown column ${columnName}`)
    }
  }
}

function _joinArgHelper(hs: types.SqlJoin | types.SqlRelation,
                        relations: RelationLookup,
                        columns: ColumnLookup,
                        catalog: types.Catalog,
                        arg: types.SqlJoin,
                        side): types.RelRelationish {
  if (hs instanceof types.SqlJoin)
    return fromJoin(hs, relations, columns, catalog)
  else if (hs instanceof types.SqlRelation)
    return fromRelation(hs, relations, columns, catalog) as types.RelRelation
  console.error(`bad join arg ${side}`, arg, "lookup:", relations)
  throw new Error("Bad join argument lhs")
}

function fromJoin(arg: types.SqlJoin,
                  relations: RelationLookup,
                  columns: ColumnLookup,
                   catalog: types.Catalog): types.RelJoin {
  const lhs = _joinArgHelper(arg.lhs, relations, columns, catalog, arg, 'left')
  const rhs = _joinArgHelper(arg.rhs, relations, columns, catalog, arg, 'right'
    )
  let cond: any = null
  if (arg.condition) {
    if (arg.condition instanceof types.SqlConditional)
      cond = fromConditional(arg.condition, relations, columns, catalog)
    else if (Array.isArray(arg.condition) && arg.condition.length === 2)
      cond = fromTargetList(arg.condition[1], relations, columns, catalog)
    else {
      console.error("bad conditional", arg, "lookup:", relations)
      throw new Error("bad conditional")
    }
  } else {
    switch (arg.joinType) {
      case "join":
      case null:
        cond = "cross"
        break
      case "leftouter":
        cond = "left"
        break
      case "rightouter":
        cond = "right"
        break
      case "fullouter":
        throw new Error("full outer join not supported")
      // case "natural" | "equi" | null:
    }
  }

  const J = new types.RelJoin(lhs, rhs, cond)
  return J
}

function fromColumn(arg: types.SqlColumn,
                    relations: RelationLookup,
                    columns: ColumnLookup,
                    catalog: types.Catalog
  ): RenameBubbleUp | ColumnValueType {
  const alias = arg.alias
  let target
  if (arg.target instanceof types.SqlColumn) {
    // column of column; either rename it or return target
    target = fromColumn(arg.target, relations, columns, catalog)
    if (!alias)
      console.warn("Why double column?")
    else if (target instanceof RenameBubbleUp) {
      console.error("Double rename; arg,target =", arg, target)
      throw new Error("Double rename not supported")
    }
  } else if (typeof(arg.target) === 'string') {
    // column based on a name
    target = columns.lookup(arg.target,
                            arg.relation || undefined,
                            arg.as || undefined)
  } else if (arg.target instanceof types.SqlLiteral) {
    target = fromLiteral(arg.target)
  } else if (arg.target instanceof types.SqlAggFunction) {
    target = fromAggFunction(arg.target, relations, columns, catalog)
  } else {
    throw new Error("Unexpected type in column")
  }

  if (alias) {
    columns.addAlias(alias, target)
    return new RenameBubbleUp(target, alias)
  }
  return target
}

function fromTargetList(targetColumns: types.SqlColumn[],
                        relationLookup: RelationLookup,
                        columnLookup: ColumnLookup,
                        catalog: types.Catalog
  ): [ColumnValueType[], RenameBubbleUp[]] {
  console.info("fromTargetList:", targetColumns)
  const renames: RenameBubbleUp[] = []
  const cols = targetColumns.map((colarg) => {
    const col = fromColumn(colarg,
                           relationLookup,
                           columnLookup,
                           catalog)
    if (col instanceof RenameBubbleUp) {
      renames.push(col)
      return col.target
    }
    return col
  })
  return [cols, renames]
}

function fromRelation(arg: types.SqlRelation,
                      relations: RelationLookup,
                      columns: ColumnLookup,
                      catalog: types.Catalog): types.RelRename | types.
                         RelRelation | types.RelJoin {
  if (typeof(arg.target) === 'string') {
    let relat
    if (relations.has(arg.target))
      relat = relations.get(arg.target)
    else if (catalog.relations.has(arg.target)) {
      relat = new types.RelRelation(arg.target)
      relations.set(arg.target, relat)
    } else {
      console.error(`Unknown relation ${arg.target}`, arg, relations)
      throw new Error(`Unknown relation ${arg.target}`)
    }

    if (arg.alias) {
      const ren = new types.RelRename(relat, arg.alias, relat)
      relations.set(arg.alias, relat)
      return ren
    }
    return relat
  } else if (arg.target instanceof types.SqlRelation) {
    const relat = fromRelation(arg.target, relations, columns, catalog) as
```

```
        types.RelRelation
    if (!arg.alias)
      return relat
    const ren = new types.RelRename(relat, arg.alias, relat)
    relations.set(arg.alias, relat)
    return ren
  } else if (arg.target instanceof types.SqlJoin) {
    const J = fromJoin(arg.target, relations, columns, catalog)
    if (!arg.alias)
      return J
    else
      throw new Error("Renaming joins not supported ")
    // const ren = new types.RelRename()
  } else {
    console.error("bad arg.target type", arg, "lookup:", relations)
    throw new Error("bad arg.target type")
  }
}

function fromRelationList(arg: types.RelationList,
                          relations: RelationLookup,
                          columns: ColumnLookup,
                          catalog: types.Catalog) {
  if (arg instanceof types.SqlRelation)
    return fromRelation(arg, relations, columns, catalog)
  else
    return fromJoin(arg, relations, columns, catalog)
}

function fromLiteral(lit: types.SqlLiteral) {
  switch (lit.literalType) {
    case 'string':
      return `'${lit.value}'`
    case 'number':
    case 'boolean':
    case 'null':
      return String(lit.value)
    default:
      throw new Error(`Unknown literal type ${lit.literalType} for ${lit.value}
          `)
  }
}

function fromAggFunction(agg: types.SqlAggFunction,
                         rels: RelationLookup,
                         cols: ColumnLookup,
                         cata: types.Catalog) {
  switch (agg.fname) {
    case 'count':
      if (agg.expr === '*' || (agg.expr as types.TargetClause).targetlist === '
          *')
        return new types.RelFunction('count', '*')
      else
        throw new Error("Counting columns not supported")
    case 'avg':
    case 'max':
    case 'min':
    case 'sum':
      if (!(agg.expr instanceof types.SqlColumn))
```

```
        throw new Error(`non-column arguments to aggregates not supported`)
      const expr = fromColumn(agg.expr, rels, cols, cata) as types.RelColumn
      return new types.RelFunction(agg.fname, expr)
    default:
      throw new Error(`Unknown aggregate function ${agg.fname}`)
  }
}

function fromOperation(arg: types.SqlOperation,
                       rels: RelationLookup,
                       cols: ColumnLookup,
                       cata: types.Catalog) {
  const lhs = _condArgHelper(arg.lhs, rels, cols, cata)
  const rhs = _condArgHelper(arg.rhs, rels, cols, cata)
  return new types.RelOperation(arg.op, lhs, rhs)
}

/* takes an Operand argument */
function _condArgHelper(hs, rels, cols, cata) {
  if (hs instanceof Array)
    return fromTargetList(hs, rels, cols, cata)[0]
  if (hs instanceof types.SqlConditional)
    return fromConditional(hs, rels, cols, cata)
  else if (hs instanceof types.SqlSelect)
    return fromSqlSelect(hs, cata)
  // Operand
  else if (hs instanceof types.SqlLiteral)
    return fromLiteral(hs)
  else if (hs instanceof types.SqlAggFunction)
    return fromAggFunction(hs, rels, cols, cata)
  else if (hs instanceof types.SqlColumn)
    return fromColumn(hs, rels, cols, cata)
  else if (hs instanceof types.SqlOperation)
    return fromOperation(hs, rels, cols, cata)
  else
    throw new Error(`Unknown conditional arg type ${hs}`)
}

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

  const tmpRhs = (arg.rhs instanceof types.SqlSelectPair)
                 ? fromSelectPair(arg.rhs, catalog)
                 : fromSqlSelect(arg.rhs, catalog)

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

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

  const rhsTarget = tmpRhs.columns

  let conditional: types.RelConditional
  if (rhsTarget.length > 1)
    conditional = rhsTarget.reduce((L, R) =>
                    new types.RelConditional(op, L, R), lhs)
  else
```

```
    conditional = new types.RelConditional(op, lhs, rhsTarget[0])

  return new BubbleUp<types.RelConditional>(conditional, tmpRhs.args)
}

function fromConditional(arg: types.SqlConditional,
                         relations: RelationLookup,
                         columns: ColumnLookup,
                         catalog: types.Catalog
  ): types.RelConditional | BubbleUp<types.RelConditional> {
  let binOp = true
  let op: types.ThetaOp
  switch (arg.operation) {
    case 'not':
    case 'isnull':
    case 'exists':
      binOp = false
      // break
    /* binary ops */
    case 'like':
    case 'between':
      throw new Error(`"${arg.operation}" condition not yet supported`)

    case 'or':
    case 'and':
    case 'in':
    case '<':
    case '>':
      op = arg.operation
      break
    case '<>':
    case '!=':
      op = 'neq'
      break
    case '<=':
      op = 'leq'
      break
    case '>=':
      op = 'geq'
      break
    case '=':
      op = 'eq'
      break
    default:
      throw new Error(`Unknown op "${arg.operation}"`)
  }
  let lhs = _condArgHelper(arg.lhs, relations, columns, catalog)
  if (lhs instanceof RenameBubbleUp) {
    lhs = lhs.target
  }

  if (op === 'in' && arg.rhs instanceof Array) {
    const rs = arg.rhs.map((R) => {
      const tcond = _condArgHelper(R, relations, columns, catalog)
      if (tcond instanceof RenameBubbleUp)
        return tcond.target
      return tcond
    })
    const cond = new types.RelConditional('in', lhs, rs)
    if (arg.not)
      throw new Error("'not' conditional is not supported")
    return cond
  }
  if (arg.rhs instanceof types.SqlSelect ||
      arg.rhs instanceof types.SqlSelectPair) {
    return _handleSubquery(arg, lhs, op, relations, columns, catalog)
  }
  if (op === 'in') {
    throw new Error("'in' argument should be array or subquery")
  }

  if (!binOp)
    throw new Error("unary operators not supported")
  let rhs = _condArgHelper(arg.rhs, relations, columns, catalog)
  if (rhs instanceof RenameBubbleUp)
    rhs = rhs.target

  const condit = new types.RelConditional(op, lhs, rhs)

  if (arg.not)
    throw new Error("'not' conditional is not supported")
  return condit
}

function fromOrderings(orderings, rels, cols, cata) {
  if (!orderings || !orderings.length)
    return null
  return orderings.map(([col, cond]) => {
    const column = fromColumn(col, rels, cols, cata)
    if (column instanceof RenameBubbleUp)
      return [column.target, cond]
    return [column, cond]
  })
}

export function fromSelectPair(selPair: types.SqlSelectPair,
                               catalog: types.Catalog) {
  const lhs = fromSqlSelect(selPair.lhs, catalog)
  let rhs
  if (selPair.rhs instanceof types.SqlSelect)
    rhs = fromSqlSelect(selPair.rhs, catalog)
  else
    rhs = fromSelectPair(selPair.rhs, catalog)

  if (lhs instanceof types.RelProjection &&
      rhs instanceof types.RelProjection) {
    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 types.RelOperation(selPair.pairing, newLhs, newRhs)
    return new types.RelProjection(newColumns, args)
  }

  const operation = new types.RelOperation(selPair.pairing, lhs, rhs)
  return operation
```

```
}

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

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

export function fromSqlSelect(select: types.SqlSelect, catalog: types.Catalog)
   {

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

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

  let targetColumns
  let renames: RenameBubbleUp[] = []
  if (select.what.targetlist === '*')
    targetColumns = '*'
  else {
    [targetColumns, renames] = fromTargetList(select.what.targetlist,
                                              relations,
                                              columns,
                                              catalog)
  }

  // const whereClause = select.where
  //     ? fromConditional(select.where, relations, columns, catalog)
  //     : null
  let whereClause: any = null
  if (select.where) {
    whereClause = fromConditional(select.where, relations, columns, catalog)
    if (whereClause instanceof BubbleUp) {
      fromClause = new types.RelJoin(fromClause, whereClause.relationish, '
          cross')
      whereClause = whereClause.realOperation as types.RelConditional
    }
  }

  if (renames.length) {
    fromClause = applyRenameBubbleUps(renames, fromClause)
  }

  const groupBy = select.groupBy
      ? fromTargetList(select.groupBy, relations, columns, catalog)
      : null

  const having = select.having
      ? fromConditional(select.having, relations, columns, catalog)
      : null

  const orderBy = fromOrderings(select.orderBy, relations, columns, catalog)

  const Rest = whereClause
      ? new types.RelRestriction(whereClause, fromClause)
      : fromClause

  const Proj = targetColumns === '*'
      ? Rest
      : new types.RelProjection(targetColumns, Rest)

  return Proj
}
```

2.3.4 src/parser/tests.ts

```

export const selectTests = [

`-- Query 2a
SELECT    S.sname
FROM      Sailors AS S, Reserves AS R
WHERE     S.sid=R.sid AND R.bid=103`,

`-- Query 2b
SELECT    S.sname
FROM      Sailors AS S, Reserves AS R, Boats AS B
WHERE     S.sid=R.sid AND R.bid=B.bid AND B.color=âĂŹredâĂŹ`,

`-- Query 2c
SELECT    sname
FROM      Sailors, Boats, Reserves
WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
Boats.color=âĂŹredâĂŹ
UNION
SELECT    sname
FROM      Sailors, Boats, Reserves
WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
Boats.color='green'`,

`-- Query 2d (invalid)
-- unescaped reserve word 'day', invalid reference 'R.rating'
SELECT    S.sname
FROM      Sailors AS S, Reserves AS R
WHERE     R.sid = S.sid AND R.bid = 100 AND R.rating > 5 AND R.day =
âĂŸ8/9/09âĂŹ`,

`-- Modified Query2d (invalid)
-- still unknown reference 'R.rating'
SELECT    S.sname
FROM      Sailors AS S, Reserves AS R
WHERE     R.sid = S.sid AND R.bid = 100 AND R.rating > 5 AND R.\`day\` =
âĂŸ8/9/09âĂŹ`,

`-- Query 2e
SELECT    sname
FROM      Sailors, Boats, Reserves
WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
Boats.color=âĂŹredâĂŹ
INTERSECT
```

```
SELECT    sname
FROM      Sailors, Boats, Reserves
WHERE     Sailors.sid=Reserves.sid AND Reserves.bid=Boats.bid AND
Boats.color=âĂŹgreenâĂŹ`,

`-- Query 2f (invalid)
-- illegal identifier '2color' of B
SELECT    S.sid
FROM      Sailors AS S,  Reserves AS R, Boats AS B
WHERE     S.sid=R.sid AND R.bid=B.bid AND B.color=âĂŸredâĂŹ
EXCEPT
SELECT    S2.sid
FROM      Sailors AS S2,  Reserves AS R2, Boats AS B2
WHERE     S2.sid=R2.sid AND R2.bid=B2.bid AND B.2color=âĂŸgreenâĂŹ`,

`-- Modified Query 2f
SELECT    S.sid
FROM      Sailors AS S,  Reserves AS R, Boats AS B
WHERE     S.sid=R.sid AND R.bid=B.bid AND B.color=âĂŸredâĂŹ
EXCEPT
SELECT    S2.sid
FROM      Sailors AS S2,  Reserves AS R2, Boats AS B2
WHERE     S2.sid=R2.sid AND R2.bid=B2.bid AND B2.color=âĂŸgreenâĂŹ`,

`-- Query 2g (invalid)
-- unknown reference 'Reserve'
SELECT    S.sname
FROM      Sailors AS S
WHERE     S.sid IN ( SELECT   R.sid
                     FROM     Reserve AS R
                     WHERE    R.bid = 103)`,

`-- Modified Query 2g
SELECT    S.sname
FROM      Sailors AS S
WHERE     S.sid IN ( SELECT   R.sid
                     FROM     Reserves AS R
                     WHERE    R.bid = 103)`,

`-- Query 2h (invalid)
-- unknown reference 'Reserve'
SELECT    S.sname
FROM      Sailors AS S
WHERE     S.sid IN ((SELECT   R.sid
                     FROM     Reserve AS R, Boats AS B
                     WHERE    R.bid = B.bid AND B.color = âĂŸredâĂŹ)
                    INTERSECT
                    (SELECT   R2.sid
                     FROM     Reserve AS R2, Boats AS B2
                     WHERE    R2.bid = B2.bid AND B2.color = âĂŸgreenâĂŹ))`,

`-- Modified Query 2h
SELECT    S.sname
FROM      Sailors AS S
WHERE     S.sid IN ((SELECT   R.sid
                     FROM     Reserves AS R, Boats AS B
                     WHERE   R.bid = B.bid AND B.color = âĂŸredâĂŹ)
                    INTERSECT
                    (SELECT   R2.sid
                     FROM     Reserves AS R2, Boats AS B2
                     WHERE    R2.bid = B2.bid AND B2.color = âĂŸgreenâĂŹ))`,

`-- Query 2i (invalid)
-- bad inner condition string, also unknown reference 'R'
SELECT   S.sname
FROM      Sailors AS S
WHERE     S.age > ( SELECT    MAX (S2.age)
                    FROM      Sailors S2
                    WHERE    R.sid = S2.rating = 10)`,

`-- Modified Query 2i
SELECT   S.sname
FROM      Sailors AS S
WHERE     S.age > ( SELECT    MAX (S2.age)
                    FROM      Sailors S2
                    WHERE     S2.rating = 10)`,

`-- Query 2j
SELECT   B.bid, Count (*) AS reservationcount
FROM      Boats B, Reserves R
WHERE     R.bid=B.bid AND B.color = âĂŸredâĂŹ
GROUP BY  B.bid`,

`-- Query 2k
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 2l (invalid)
-- typo "SLECT", misuse of nonstandard 'contains' WHERE predicate, 'Sname'
SELECT    Sname
FROM      Sailors
WHERE     Sailor.sid IN (SELECT   Reserves.bid, Reserves.sid
                         FROM     Reserves
                         CONTAINS
                                  (SLECT Boats.bid
                                   FROM  Boats
                                   WHERE Boats.name  =  âĂŸinterlakeâĂŹ) )`,

`-- Modified Query 2l (invalid, system-specific)
-- unknown reference 'Sname'
SELECT    Sname
FROM      Sailors
WHERE     Sailor.sid IN (SELECT   Reserves.bid, Reserves.sid
                         FROM     Reserves
                         WHERE    EXISTS (
                                  SELECT Boats.bid
                                  FROM  Boats
                                  WHERE Boats.name  =  âĂŸinterlakeâĂŹ
                                        AND Boats.bid = Reserves.bid ) )`,

`-- Query 2m (invalid)
-- Bad TargetList
SELECT    S.rating, Ave (S.age) As average
FROM      Sailors S
WHERE     S.age > 18
```

```
GROUP BY  S.rating
HAVING    Count (*) > 1`,

`-- Modified Query 2m
SELECT    S.rating, Avg (S.age) As average
FROM      Sailors S
WHERE     S.age > 18
GROUP BY  S.rating
HAVING    Count (*) > 1`
]
```

2.3.5 src/parser/types.ts

```

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 const REL_RESTRICTION_TYPE = "restriction"
export const REL_PROJECTION_TYPE  = "projection"
export const REL_RENAME_TYPE      = "rename"

export const REL_RELATION_TYPE    = "relrelation"
export const REL_COLUMN_TYPE      = "relcolumn"
export const REL_CONDITIONAL_TYPE = "relconditional"
export const REL_JOIN_TYPE        = "reljoin"
export const REL_FUNCTION_TYPE    = "relfunt"
export const REL_OPERATION_TYPE   = "relop"

/**
 * IFF rhs is non-empty, run reduce using f on rhs initialized by lhs.
 * Else return lhs
 */
export function reduceIfRHS(lhs: any, rhs: any[], f: (L, R) => any) {
  if (rhs.length)
    return rhs.reduce(f, lhs)
  return lhs
}

export class Catalog {

  static fromParse(relations: Array<[string, Array<[string, string]>]>) {
    const rels = new Map()
    relations.forEach((ele) => {
      const [tname, cols] = ele
      const columnMap = new Map()
      cols.forEach((col) => {
        columnMap.set(col[0], new Column(col[0], col[1]))
      })
      rels.set(tname, new Relation(tname, columnMap))
    })
    return new Catalog(rels)
  }

  relations: Map<string, Relation>

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

export class Relation {
  name: string
  columns: Map<string, Column>

  constructor(name: string, columns: Map<string, Column>) {
    this.name = name
    this.columns = columns
  }
}

export class Column {
  name: string
  typ: string

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

export type JOINSTRING = "join"        // "," | "JOIN" | "CROSS JOIN"
                       | "equi"        // "INNER JOIN" | "JOIN ... USING"
                       | "natural"     // "NATURAL JOIN"
                       | "leftouter"   // "LEFT [OUTER] JOIN"
                       | "rightouter"  // "RIGHT [OUTER] JOIN"
                       | "fullouter"   // "FULL [OUTER] JOIN"

type OrderingCondition = "asc" | "desc" | "<" | ">"
type Ordering = [SqlColumn, OrderingCondition]

export type RelationList = SqlRelation | SqlJoin
type TargetList = SqlColumn[]

export interface TargetClause {
  type: "targetclause"
  spec: "distinct" | "all" | null
  targetlist: "*" | TargetList
}

export class SqlLiteral {
  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 SqlSelectish = SqlSelect | SqlSelectPair
export type PairingString = 'union' | 'intersect' | 'except'
export type PairingCondition = 'all' | 'distinct' | null

export class SqlSelectPair {
  readonly type = SELECTPAIR_TYPE
  pairing: PairingString
  condition: PairingCondition
  lhs: SqlSelect
  rhs: SqlSelectish

  constructor(pairing: PairingString,
              condition: PairingCondition,
              lhs: SqlSelect,
              rhs: SqlSelectish) {
    this.pairing = pairing
    this.condition = condition || null
    this.lhs = lhs
    this.rhs = rhs
  }
}

export class SqlSelect {
  readonly SELECTCLAUSE_TYPE
  what: TargetClause
  from: RelationList
  where: SqlConditional | null
  groupBy: TargetList | null
  having: SqlConditional | null
  orderBy: Ordering[] | null

  constructor(what: TargetClause,
              from: RelationList,
              where: SqlConditional | null,
              groupBy: TargetList | null,
              having: SqlConditional | null,
              orderBy: Ordering[] | null) {
    this.what = what
    this.from = from
    this.where = where
    this.groupBy = groupBy
    this.having = having
    this.orderBy = orderBy
  }
}

export type SqlOperandType = SqlLiteral | SqlAggFunction | SqlColumn |
                      SqlOperation | string

export class SqlColumn {
  readonly type = COLUMN_TYPE
  relation: string | null
  target: SqlOperandType
  as: string | null
  alias: string | null

  constructor(relation: string | null,
              target: SqlOperandType,
              As: string | null = null,
              alias: string | null = null) {
    this.relation = relation
    this.target = target
    this.as = As || null
    this.alias = alias || null
  }
}

export class SqlJoin {
  readonly type = JOIN_TYPE
  joinType: JOINSTRING
  condition: SqlConditional | ['using', TargetList] | null
  lhs: SqlJoin | SqlRelation
  rhs: SqlJoin | SqlRelation

  constructor(lhs: SqlJoin | SqlRelation,
              rhs: SqlJoin | SqlRelation,
              joinType: JOINSTRING = 'join',
              condition: SqlConditional | ['using', TargetList] | null = null
  ) {
    this.lhs = lhs
    this.rhs = rhs
    this.joinType = joinType || 'join'
    this.condition = condition || null
  }
}

export class SqlRelation {
  readonly type = RELATION_TYPE
  target: SqlRelation | SqlJoin | string
  alias: string | null

  constructor(target: SqlRelation | SqlJoin | string,
              alias: string | null = null) {
    this.target = target
    this.alias = alias || null
  }
}

export type SqlConditionalOp = 'or' | 'and' | 'not' | 'in' | 'exists' | 'like'
  |
                               'between' | 'isnull' | '<>' | 'contains' |
                               '<=' | '>=' | '=' | '<' | '>' | '!='

export class SqlConditional {
  readonly type = CONDITIONAL_TYPE
  operation: SqlConditionalOp
  lhs: SqlConditional | SqlOperandType
  rhs: SqlConditional | SqlOperandType | null
  not: boolean

  constructor(operation: SqlConditionalOp,
              lhs: SqlConditional | SqlOperandType,
              rhs: SqlConditional | SqlOperandType | null = null,
              not: boolean = false) {
    if (operation === 'in' && lhs instanceof Array && lhs.length === 1)
```

```
      lhs = lhs[0]
    this.operation = operation
    this.lhs = lhs
    this.rhs = rhs || null
    this.not = not
  }
}

export type AggFuncName = 'avg' | 'count' | 'max' | 'min' | 'sum'

export class SqlAggFunction {
  readonly type = AGGFUNCTION_TYPE
  fname: AggFuncName
  expr: SqlOperandType | TargetClause

  constructor(fname: AggFuncName, expr: SqlOperandType | TargetClause) {
    this.fname = fname
    this.expr = expr
  }
}

export type SqlOperationOps = '||' | '+' | '-' | '*' | '/'

export class SqlOperation {
  readonly type = OPERATION_TYPE
  op: SqlOperationOps
  lhs: SqlOperandType
  rhs: SqlOperandType

  constructor(op: SqlOperationOps, lhs: SqlOperandType, rhs: SqlOperandType) {
    this.op = op
    this.lhs = lhs
    this.rhs = rhs
  }
}

/*** RELATIONAL ALGEBRA ***/
// literals are strings

export type RelRelationish = RelRelation | RelJoin
export type RelOperandType = RelOperation | string | RelColumn

export class RelOperation {
  readonly type = REL_OPERATION_TYPE
  op: SqlOperationOps | 'union' | 'intersect' | 'except'
  lhs: RelOperandType | HighLevelRelationish
  rhs: RelOperandType | HighLevelRelationish

  constructor(op: SqlOperationOps | 'union' | 'intersect' | 'except',
              lhs: RelOperandType | HighLevelRelationish,
              rhs: RelOperandType | HighLevelRelationish) {
    this.op = op
    this.lhs = lhs
    this.rhs = rhs
  }
}

type ColumnValueType = Column | RelFunction | string

```

```
export class RelColumn {
  readonly type = REL_COLUMN_TYPE
  relation: RelRelation | null
  target: ColumnValueType
  as: string | null

  constructor(relation: RelRelation | null,
              target: ColumnValueType,
              As: string | null = null) {
    this.relation = relation
    this.target = target
    this.as = As || null
  }

  alias(alias?: string) {
    if (!alias)
      return this
    return new RelColumn(this.relation, this.target, alias)
  }
}

export class RelFunction {
  readonly type = REL_FUNCTION_TYPE
  fname: AggFuncName
  expr: '*' | RelColumn // TODO: support correct args

  constructor(fname: AggFuncName, expr: '*' | RelColumn) {
    this.fname = fname
    this.expr = expr
  }
}

export type ThetaOp = 'eq' | 'neq' | 'leq' | 'geq' | '<' | '>' | 'and' | 'or' |
                      'in'

export class RelConditional {
  readonly type = REL_CONDITIONAL_TYPE
  operation: ThetaOp
  lhs: RelOperandType | RelConditional
  rhs: RelOperandType | RelConditional | RelOperandType[]

  constructor(op: ThetaOp, lhs: RelOperandType | RelConditional,
              rhs: RelOperandType | RelConditional | RelOperandType[]) {
    this.operation = op
    this.lhs = lhs
    this.rhs = rhs
  }
}

export type HighLevelRelationish = RelRelationish | RelRestriction |
    RelProjection | RelRename | RelOperation

export class RelRestriction {
  readonly type = REL_RESTRICTION_TYPE
  conditions: RelConditional
  args: HighLevelRelationish

  constructor(conditions: RelConditional, args: HighLevelRelationish) {
    this.conditions = conditions
    this.args = args
  }
}

export class RelProjection {
  readonly type = REL_PROJECTION_TYPE
  columns: RelColumn[]
  args: HighLevelRelationish

  constructor(columns: RelColumn[], args: HighLevelRelationish) {
    this.columns = columns
    this.args = args
  }
}

type _RelRenameInputType = RelRelation | RelColumn | RelFunction |
                           RelRename | string

export class RelRename {
  readonly type = REL_RENAME_TYPE
  input: _RelRenameInputType
  output: string
  args: HighLevelRelationish

  constructor(input: _RelRenameInputType,
              output: string,
              args: HighLevelRelationish) {
    this.input = input
    this.output = output
    this.args = args
  }
}

export class RelRelation {
  readonly type = REL_RELATION_TYPE
  name: string

  constructor(name: string) {
    this.name = name
  }
}

export type RelJoinCond = "cross" | "left" | "right" | RelConditional

// cross
// natural (no condition)
// theta join (with condition)
// semi (left and right)
export class RelJoin {
  readonly type = REL_JOIN_TYPE
  lhs: HighLevelRelationish
  rhs: HighLevelRelationish
  condition: RelJoinCond

  constructor(lhs: HighLevelRelationish,
              rhs: HighLevelRelationish,
              cond: RelJoinCond) {
    this.lhs = lhs
    this.rhs = rhs
    this.condition = cond
  }
}
```

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
  = Selectish

Selectish
  = SelectPair
  / Select


SelectPair
  = lhs:Select __
    pairing:$( "UNION"i / "INTERSECT"i / "EXCEPT"i ) __
    spec:( "ALL"i __ / "DISTINCT"i __ )?
    rhs:( Selectish )
  {
    return new SqlSelectPair(pairing.toLowerCase(),
                             spec && spec[0].toLowerCase(),
                             lhs,
                             rhs)
  }

Select
  = "SELECT"i __ what:TargetClause __
    "FROM"i   __ from:FromClause
    where:(   __ "WHERE"i  __           WhereClause )?
    groupBy:( __ "GROUP"i  __ "BY"i __ GroupByClause )?
    having:(  __ "HAVING"i __           HavingClause )?
    orderBy:( __ "ORDER"i  __ "BY"i __ OrderByClause )?
  {
    return new SqlSelect(what, from, where && where[3],groupBy && groupBy[5],
                         having && having[3], orderBy && orderBy[5])
  }
  / "(" _ sel:Select _ ")" { return sel }
```

```

TargetClause
  = spec:$( "DISTINCT"i __ / "ALL"i __ )?
    target:(
      "*"
      / TargetList
    )
  { return {
      'type': TARGETCLAUSE_TYPE,
      'specifier': spec ? spec.toLowerCase() : null,
      'targetlist': target
    }
  }

FromClause
  = from:RelationList

WhereClause
  = where:Condition

GroupByClause
  = groupBy:TargetList

HavingClause
  = having:Condition

OrderByClause
  = lhs:Ordering rhs:( _ "," _ Ordering )*
  { return rhs.reduce((result, element) => result.concat(element[3]), [lhs]) }

Ordering
  = expr:Operand
    cond:(
        __ "ASC"i { return 'asc' }
      / __ "DESC"i { return 'desc' }
      / __ "USING"i _ op:$( "<" / ">" ) { return op }
    )?

RelationList
  = item1:RelationItem _ "," _ items:RelationList
    { return new SqlJoin(item1, items) }
    / Join
    / RelationItem

RelationItem "RelationItem"
  = item:RelationThing __ ( "AS"i __ )? alias:Name
  { return new SqlRelation(item, alias) }
  / RelationThing

RelationThing
  = "(" _ list:RelationList _ ")"
  { return list }
  / "(" _ join:Join _ ")"
  { return join }
  / tableName:Name
  { return new SqlRelation(tableName) }

Join
  = item1:RelationItem __
```

```
    jtype:JoinType __
    item2:RelationItem
    jcond:(
      __ "ON"i
      __ expr:Condition
      { return expr }
      / __ "USING"i _
        "(" _ list:TargetList _ ")"
        { return ['using', list] }
    )?
  { return new SqlJoin(item1, item2, jtype, jcond) }

TargetList
  = item1:TargetItem _ "," _ items:TargetList
    { return [item1].concat(items) }
    / item:TargetItem
    { return [item] }

TargetItem "TargetItem"
  = table:Name ".*"
  { return new SqlColumn(table, '*', `${table}.*`, null) }
  / op:Operand __ "AS"i __ alias:Name
  { return new SqlColumn(null, op, alias, alias )}
  / op:Operand __ alias:Name
  { return new SqlColumn(null, op, alias, alias )}
  / op:Operand _ "=" _ alias:Name
  { return new SqlColumn(null, op, alias, alias) }
  / op:Operand
  { return (op instanceof SqlColumn) ? op : new SqlColumn(null, op) }

Condition "Condition"
  = lhs:AndCondition rhs:( __ "OR"i __ Condition )?
  { return rhs ? new SqlConditional('or', lhs, rhs[3]) : lhs }

AndCondition
  = lhs:InnerCondition rhs:( __ "AND"i __ AndCondition )?
  { return rhs ? new SqlConditional('and', lhs, rhs[3]) : lhs }

InnerCondition
  = ( ConditionContains
    / ConditionComp
    / ConditionIn
    / ConditionExists
    / ConditionLike
    / ConditionBetween
    / ConditionNull
//    / Operand
  )
  / "NOT"i __ expr:Condition
  { return new SqlConditional('not', expr) }
  / "(" _ expr:Condition _ ")"
  { return expr }

ConditionContains "Conditional-Contains"
  // based on Transact-SQL
  = "CONTAINS" _
    "(" _
      lhs:(
        Operand
        / "(" _ ops:OperandList _ ")"
        { return ops }
      )
      rhs:SQStringLiteral
    ")"
  { return new SqlConditional('contains', lhs, rhs) }

ConditionComp "Conditional-Comparison"
  = lhs:Operand _ cmp:Compare _ rhs:Operand
  { return new SqlConditional(cmp, lhs, rhs) }

ConditionIn
  = lhs_op:Operand __
    not:( "NOT"i __ )?
    "IN"i _
    "(" _
      rhs_ops:( Selectish / OperandList ) _
    ")"
  { return new SqlConditional('in', lhs_op, rhs_ops, not) }

ConditionExists
  = "EXISTS"i _
    "(" _ subquery:Selectish _ ")"
  { return new SqlConditional('exists', subquery) }

ConditionLike
  = lhs_op:Operand __
    not:( "NOT"i __ )?
    "LIKE"i __
    rhs_op:Operand
  { return new SqlConditional('like', lhs_op, rhs_op, not) }

ConditionBetween
  = lhs_op:Operand __
    not:( "NOT"i __ )?
    "BETWEEN"i
    rhs:(
      __
      rhs_op1:Operand __
      "AND"i __
      rhs_op2:Operand
      { return [rhs_op1, rhs_op2] }
      / _
        "(" _
          rhs_op1:Operand __
          "AND"i __
          rhs_op2:Operand
        ")"
        { return [rhs_op1, rhs_op2] }
    )
  { return new SqlConditional('between', lhs_op, rhs, not) }

ConditionNull
  = lhs:Operand __ "IS"i __
    not:( "NOT"i __ )?
    NullLiteral
  { return new SqlConditional('isnull', lhs, null, not) }

Term
```

```
  = Literal
    / AggFunction
    / "(" _ op:Operand _ ")" { return op }
    / ColumnRef

ColumnRef
  = tbl:( table:Name "." )? column:Name
    { return new SqlColumn(tbl && tbl[0],
                           column,
                           tbl ? `${tbl[0]}.${column}` : column
                         ) }

AggFunction "aggregate function"
  = AggFunctionAvg
    / AggFunctionCount
    / AggFunctionMax
    / AggFunctionMin
    / AggFunctionSum

AggFunctionAvg
  = "AVG"i _
    "(" _ term:Term _ ")"
  { return new SqlAggFunction("avg", term) }

AggFunctionCount
  = "COUNT"i _
    "(" _
      targ:TargetClause _
    ")"
  { return new SqlAggFunction("count", targ) }

AggFunctionMax
  = "MAX"i _
    "(" _
      term:Term _
    ")"
  { return new SqlAggFunction("max", term) }

AggFunctionMin
  = "MIN"i _
    "(" _
      term:Term _
    ")"
  { return new SqlAggFunction("min", term) }

AggFunctionSum
  = "SUM"i _
    "(" _
      term:Term _
    ")"
  { return new SqlAggFunction("sum", term) }

/***** PRIMITIVES *****/

Name
  = DQStringLiteral
    / BTStringLiteral
    / !ReservedWord id:Ident {return id }

```

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

OperandList
  = lhs:Operand
    rhs:( _ "," _ Operand )*
  {
    if (rhs.length)
      return rhs.reduce((result, element) => result.concat(element[3]), [lhs])
    else
      return lhs
  }

Operand // Summand | makeOperation
  = lhs:Summand
    rhs:( _ "||" _ Summand ) *
  { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation("||",
                                                               lh, rh[3])) }
  / Selectish

Summand // Factor | makeOperation
  = lhs:Factor
    rhs:( _ ("+" / "-") _ Factor ) *
  { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation(rh[1],
                                                               lh, rh[3])) }

Factor // literal | function | Operand | column | makeOperation
  = lhs:Term
    rhs:( _ ("*" / "/") _ Term ) *
  { return reduceIfRHS(lhs, rhs, (lh, rh) => new SqlOperation(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:$( "'" ( [^'] / "''" )* "'" !SQStringLiteral )
  { return new SqlLiteral('string', lit.slice(1, -1)) }
  / lit:$( ("âĂŸ"/"âĂŹ") ( [^âĂŹ] )* "âĂŹ" ) // fancy single-quote
  { return new SqlLiteral('string', lit.slice(1, -1)) }

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

NumericLiteral "number"
  = IntegerLiteral
    / DecimalLiteral

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

DecimalLiteral "decimal"
  = value:$( "-"? [0-9]+ "." [0-9]+ )
  { return new SqlLiteral('number', parseFloat(value)) }

NullLiteral "null"
  = "NULL"i
  { return new SqlLiteral('null', null) }

BooleanLiteral "boolean"
  = TruePrim
    / FalsePrim

TruePrim
  = "TRUE"i
  { return new SqlLiteral('boolean', true) }

FalsePrim
  = "FALSE"i
  { return new SqlLiteral('boolean', false) }

_ "OptWhitespace"
  = WS* (Comment WS*)* {}

__ "ReqWhitespace"
  = WS+ (Comment WS*)* {}

WS
  = [ \t\n]

Comment "Comment"
```

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

/** SQL2008 reserved words.
    In alphabetical order but not always lexical order,
      as there is no backtracking in PEG.js, e.g. for
        "IN" / "INT" / "INTERSECT" / "INTERSECTION"
      only "IN" is reachable.
 **/
ReservedWord
  = $("ABS"i / "ALL"i / "ALLOCATE"i / "ALTER"i / "AND"i / "ANY"i / "ARE"i /
        "ARRAY_AGG"i / "ARRAY"i / "ASENSITIVE"i / "ASYMMETRIC"i / "AS"i /
        "ATOMIC"i / "AT"i / "AUTHORIZATION"i / "AVG"i
      / "BEGIN"i / "BETWEEN"i / "BIGINT"i / "BINARY"i / "BLOB"i / "BOOLEAN"i /
        "BOTH"i / "BY"i
      / "CALLED"i / "CALL"i / "CARDINALITY"i / "CASCADED"i / "CASE"i / "CAST"i /
        "CEILING"i / "CEIL"i / "CHARACTER_LENGTH"i / "CHAR_LENGTH"i /
        "CHARACTER"i / "CHAR"i / "CHECK"i / "CLOB"i / "CLOSE"i / "COALESCE"i /
        "COLLATE"i / "COLLECT"i / "COLUMN"i / "COMMIT"i / "CONDITION"i /
        "CONNECT"i / "CONSTRAINT"i / "CONVERT"i / "CORRESPONDING"i / "CORR"i /
        "COUNT"i / "COVAR_POP"i / "COVAR_SAMP"i / "CREATE"i / "CROSS"i /
        "CUBE"i / "CUME_DIST"i / "CURRENT_CATALOG"i / "CURRENT_DATE"i /
        "CURRENT_DEFAULT_TRANSFORM_GROUP"i / "CURRENT_PATH"i / "CURRENT_ROLE"i /
        "CURRENT_SCHEMA"i / "CURRENT_TIMESTAMP"i / "CURRENT_TIME"i /
        "CURRENT_TRANSFORM_GROUP_FOR_TYPE"i / "CURRENT_USER"i / "CURRENT"i /
        "CURSOR"i / "CYCLE"i
      / "DATALINK"i / "DATE"i / "DAY"i / "DEALLOCATE"i / "DECIMAL"i /
        "DECLARE"i / "DEC"i / "DEFAULT"i / "DELETE"i / "DENSE_RANK"i /
        "DEREF"i / "DESCRIBE"i / "DETERMINISTIC"i / "DISCONNECT"i /
        "DISTINCT"i / "DLNEWCOPY"i / "DLPREVIOUSCOPY"i / "DLURLCOMPLETE"i /
        "DLURLCOMPLETEONLY"i / "DLURLCOMPLETEWRITE"i / "DLURLPATHONLY"i /
        "DLURLPATHWRITE"i / "DLURLPATH"i / "DLURLSCHEME"i / "DLURLSERVER"i /
        "DLVALUE"i / "DOUBLE"i / "DROP"i / "DYNAMIC"i
      / "EACH"i / "ELEMENT"i / "ELSE"i / "END-EXEC"i / "END"i / "ESCAPE"i /
        "EVERY"i / "EXCEPT"i / "EXECUTE"i / "EXEC"i / "EXISTS"i / "EXP"i /
        "EXTERNAL"i / "EXTRACT"i
      / "FALSE"i / "FETCH"i / "FILTER"i / "FIRST_VALUE"i / "FLOAT"i / "FLOOR"i /
        "FOREIGN"i / "FOR"i / "FREE"i / "FROM"i / "FULL"i / "FUNCTION"i /
        "FUSION"i
      / "GET"i / "GLOBAL"i / "GRANT"i / "GROUPING"i / "GROUP"i
      / "HAVING"i / "HOLD"i / "HOUR"i
      / "IDENTITY"i / "IMPORT"i / "INDICATOR"i / "INNER"i / "INOUT"i /
        "INSENSITIVE"i / "INSERT"i / "INTEGER"i / "INTERSECTION"i /
        "INTERSECT"i / "INTERVAL"i / "INTO"i / "INT"i / "IN"i / "IS"i
      / "JOIN"i
      / "LAG"i / "LANGUAGE"i / "LARGE"i / "LAST_VALUE"i / "LATERAL"i /
        "LEADING"i / "LEAD"i / "LEFT"i / "LIKE_REGEX"i / "LIKE"i / "LN"i /
        "LOCALTIMESTAMP"i / "LOCAL"i / "LOCALTIME"i / "LOWER"i
      / "MATCH"i / "MAX_CARDINALITY"i / "MAX"i / "MEMBER"i / "MERGE"i /
        "METHOD"i / "MINUTE"i / "MIN"i / "MODIFIES"i / "MODULE"i / "MOD"i /
        "MONTH"i / "MULTISET"i
      / "NATIONAL"i / "NATURAL"i / "NCHAR"i / "NCLOB"i / "NEW"i / "NONE"i /
        "NORMALIZE"i / "NOT"i / "NO"i / "NTH_VALUE"i / "NTILE"i / "NULLIF"i /
        "NULL"i / "NUMERIC"i
      / "OCCURRENCES_REGEX"i / "OCTET_LENGTH"i / "OFFSET"i / "OF"i / "OLD"i /
        "ONLY"i / "ON"i / "OPEN"i / "ORDER"i / "OR"i / "OUTER"i / "OUT"i /
        "OVERLAPS"i / "OVERLAY"i / "OVER"i
      / "PARAMETER"i / "PARTITION"i / "PERCENTILE_CONT"i / "PERCENTILE_DISC"i /
        "PERCENT_RANK"i / "POSITION_REGEX"i / "POSITION"i / "POWER"i /
        "PRECISION"i / "PREPARE"i / "PRIMARY"i / "PROCEDURE"i
      / "RANGE"i / "RANK"i / "READS"i / "REAL"i / "RECURSIVE"i / "REFERENCES"i
         /
        "REFERENCING"i / "REF"i / "REGR_AVGX"i / "REGR_AVGY"i / "REGR_COUNT"i /
        "REGR_INTERCEPT"i / "REGR_R2"i / "REGR_SLOPE"i / "REGR_SXX"i /
        "REGR_SXY"i / "REGR_SYY"i / "RELEASE"i / "RESULT"i / "RETURNS"i /
        "RETURN"i / "REVOKE"i / "RIGHT"i / "ROLLBACK"i / "ROLLUP"i / "ROWS"i /
        "ROW_NUMBER"i / "ROW"i
      / "SAVEPOINT"i / "SCOPE"i / "SCROLL"i / "SEARCH"i / "SECOND"i /
        "SELECT"i / "SENSITIVE"i / "SESSION_USER"i / "SET"i / "SIMILAR"i /
        "SMALLINT"i / "SOME"i / "SPECIFICTYPE"i / "SPECIFIC"i /
        "SQLEXCEPTION"i / "SQLSTATE"i / "SQLWARNING"i / "SQL"i / "SQRT"i /
        "START"i / "STATIC"i / "STDDEV_POP"i / "STDDEV_SAMP"i / "SUBMULTISET"i
           /
        "SUBSTRING_REGEX"i / "SUBSTRING"i / "SUM"i / "SYMMETRIC"i /
        "SYSTEM_USER"i / "SYSTEM"i
      / "TABLESAMPLE"i / "TABLE"i / "THEN"i / "TIMESTAMP"i / "TIMEZONE_HOUR"i /
        "TIMEZONE_MINUTE"i / "TIME"i / "TO"i / "TRAILING"i /
        "TRANSLATE_REGEX"i / "TRANSLATE"i / "TRANSLATION"i / "TREAT"i /
        "TRIGGER"i / "TRIM_ARRAY"i / "TRIM"i / "TRUE"i / "TRUNCATE"i
      / "UESCAPE"i / "UNION"i / "UNIQUE"i / "UNKNOWN"i / "UNNEST"i / "UPDATE"i
         /
        "UPPER"i / "USER"i / "USING"i
      / "VALUES"i / "VALUE"i / "VARBINARY"i / "VARCHAR"i / "VARYING"i /
        "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/query_tree

2.5.1 src/query_tree/node.ts

```
import {Operation} from './operation'

export default
class Node {
  operation: Operation
  children: Node[] = []

  constructor(operation: Operation) {
    this.operation = operation
  }

  addNode(node: Node) {
    this.children.push(node)
  }
}
```

2.5.2 src/query_tree/operation.ts

```

export class Operation {
  name: string
  //TODO: better define this type
  arguments: string[] = []

  constructor(name: string) {
    this.name = name
  }

  addArgument(arg: string) {
    this.arguments.push(arg)
  }
}

export class Projection extends Operation {
  constructor() {
    super("Project")
  }

  addTarget(data) {
    let { relation, target, alias} = data
    let arg: string = `${relation} ${target}`
    if(alias)
      arg += ` as ${alias}`

    this.addArgument(arg)
  }
}

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
    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.value) arg = data.value

    return arg
  }
}
```

2.5.3 src/query_tree/parse.ts

```
// tslint:disable
import Node from './node'
import {Operation, Projection, From, Where} from './operation'

// convert json produced by peg to Tree
export default function parseSQLToTree(sql): Node {
  // TODO: fix order of tree hierarchy

  let projectArgs = sql[0].what.targetlist
  let op = new Projection()
  projectArgs.forEach(arg => op.addTarget(arg))
  let root = new Node(op)

  let fromArgs = sql[0].from
  let from = new From()
  from.addTarget(fromArgs)
  let fromNode = new Node(from)
  root.addNode(fromNode)

  let whereArgs = sql[0].where
  let where = new Where()
  where.addTarget(whereArgs)
  let whereNode = new Node(where)
  fromNode.addNode(whereNode)

  return root
}
```