

Dokumentace k semestrální práci z předmětu
KIV/DBM2

Bc. Tomáš Vyleta

21.11.2020



(a)



(b)



(c)

```
alasql('ATTACH FILESTORAGE DATABASE test2("./js/file.json");USE
test2;',function () {
  console.log(alasql.databases);
});

const myDatabase1 = new alasql.Database();
myDatabase1.exec('CREATE TABLE table1 (number INT)');
myDatabase1.exec('INSERT INTO table1 (number) VALUES (1), (2), (3),
(4), (5)');

myDatabase1.exec('SELECT * FROM table1', [],
  function (res) {
    console.log(res);
  }
);

const ceskaMesta = [
  {name: 'Prague', population: 1324277},
  {name: 'Brno', population: 381346},
  {name: 'Ostrava', population: 287968},
  {name: 'Plzen', population: 174842}
];

alasql.databases.myDatabase2.tables.cities.data = ceskaMesta;
vypisVsechnaMesta();
myDatabase2.exec("INSERT INTO cities VALUES ('Rome',2863223),('
Paris ',2249975),('Berlin',3517424),('Madrid',3041579)");
vypisVsechnaMesta();

const allCities = myDatabase2.exec("SELECT * FROM cities WHERE
population < 3500000 ORDER BY population DESC");
console.log(allCities);

myDatabase2.exec('UPDATE cities SET population = population * 1.5
WHERE name LIKE "A%"');
console.log(alasql.databases.myDatabase2);
```

Obsah

1	Knihovna AlaSQL	4
1.1	Infobox	5
1.2	Užitečné odkazy	5
2	Stack	5
3	Instalace	5
4	Použití	5
4.1	Pomocí funkce	6
4.1.1	NodeJS	6
4.2	Nová databáze	6
4.3	Nová tabulka	6
4.3.1	Datové typy	7
4.4	Manipulace dat v tabulce	7
4.5	Načítání souborů	8
5	Synchronní/Asynchronní přístup	9
5.1	HTML	9
6	SQL dotazy	9
6.1	Nejlépe funguje s JS	9
7	Datasety	10
7.1	Rohlík.cz	10
7.2	Starcraft	10
8	Důležitá je cesta, ne cíl	11
8.1	Rohlík.cz	11
8.1.1	Jaké je nejbližší okno s volnou kapacitou?	11
8.1.2	Kolik volných časových intervalů je k dispozici?	12
8.1.3	Jaké je nejbližší nejlevnější okno s volnou kapacitou?	13
8.2	Starcraft	14
8.2.1	Kolik hráčů je v jednotlivých divizích/úrovních?	14
8.2.2	Jsou konzistentní počty hráčů mezi JSON typ A a B?	15
8.2.3	Seřazení hráčů v rámci jednoho ladderu.	16
8.2.4	Kolik hráčů odehrálo v poslední době nějakou hru?	18
8.2.5	Jací hráči jsou v jednom ladderu vícekrát a za jaké rasy?	19
8.2.6	Jací hráči přibyli/ubyli v daných ladderech?	20
8.2.7	Jací hráči vybočují z hranic divizí?	21
8.2.8	Kolik zástupců mají týmy v jednotlivých úrovních?	22
8.3	Objevené zajímavé funkce	24

9	Chyby	25
9.1	Procházení zanořených objektů	25
9.2	Nefunkční	26
9.3	JOIN	26
10	Popis přiložených souborů	26
11	Porovnání s SQL	26
12	Na závěr	27

Úvod

Problémem či zadáním semestrální práce je nalézt a popsat knihovnu pro zpracování množiny souborů ve formátu JSON, které většinou vrací endpointy REST API. Využít ji na úrovni jednoho programu (bez nutnosti vkládání dat do jiných SŘBD pro zpracování objektových souborů). Knihovna by měla být schopna poskytovat funkce obsažené v jazyce SQL jako FROM, GROUP BY, JOIN, atd. Cíle práce jsou:

- Nalezněte existující metody dotazováním se nad množinou JSON souborů.
- Popište rozsah nabízených operací ve vybraném jazyce/nástroji.
- Analyzujte možnosti jazyka/nástroje v kontrastu s klasickou relační databází resp. SQL.

1 Knihovna AlaSQL

AlaSQL (čti à la SQL) je *open source* SQL databáze pro JavaScript, operující na straně klienta. Implementuje mnoho funkcí ze čtvrté verze jazyka SQL (SQL:1999) a také některé funkce navíc pro snazší manipulaci s NoSQL a grafovými sítěmi. Také podporuje asynchronní volání pomocí metody promise. Knihovnu lze využít pro webové aplikace, aplikace založené na Node.js nebo v mobilních aplikacích.

Podporuje import/export formátů, jako jsou např. tabulky programu Excel (.xls), CSV - Comma-separated values (.csv), již zmiňovaný JSON - JavaScript Object Notation (.json), TAB - Tab Separated Data File (.tab) a import/export databází IndexedDB, LocalStorage a SQLite.



Obrázek 1: Logo AlaSQL

1.1 Infobox

Jazyk	JavaScript
Stabilní verze	v0.6.5
Licence	MIT Licence
Vyvíjeno od	2014
GitHub hvězd	5 300
Commity	2-3/týden
Návštěvnost	2 mil./měsíc

1.2 Užitečné odkazy

- WEB: <http://alasql.org>
- Github: <https://github.com/agershun/alasql>
- Wiki (Docs): <https://github.com/agershun/alasql/wiki>

2 Stack

Pro práci s knihovnou jsem zvolil *single page* aplikaci, kde všechny dotazy běží na klientovi s podporou JavaScriptu. Aplikace neběží na žádném serveru, tudíž nelze načítat soubory pomocí běžných metod, proto jsou všechna data uložena jako proměnné v JavaScriptu. Pokud ale pracujete s knihovnou v prostředí Node.JS, můžete soubory načítat pomocí *File system* modulu nebo pokud Node.JS nevyužíváte a aplikace vám běží na serveru, tak je zde možnost načítat soubory např. pomocí JQuery (asynchronně) nebo pomocí dalších externích knihoven a frameworků. Alasql dokáže načítat soubory asynchronně, ale metody kterými to jde, mají divné ukládání dat viz kapitola [načítání souborů](#).

3 Instalace

Knihovna je dostupná v několika JavaScriptových správčích balíčků (package management) kterými jsou npm, Bower a Meteor nebo si ji můžete přímo naklonovat/stáhnout z oficiálního GitHub repozitáře.

Pokud nechcete stahovat knihovnu na fyzické úložiště, nejjednodušší varianta je získat AlaSQL do své aplikace pomocí cloudové služby cdnjs.com, která je založena na principu CDN (Content Delivery Network). Stačí tedy do našeho projektu vložit odkaz na tuto knihovnu, např. do hlavičky HTML souboru:

```
<script src="//cdn.jsdelivr.net/alasql/0.2/alasql.min.js"></script>
```

4 Použití

Po úspěšném nainstalování/nainportování knihovny jsme schopni ji začít používat. Nyní je několik možností jak s knihovnou pracovat. Chtěl bych zdůraznit,

že knihovna vrací vždy pole výsledků nebo JavaScriptový objekt, tabulku pouze v případě, pokud zavoláme metodu tomu určenou.

4.1 Pomocí funkce

Všechny dotazy (queries) se vkládají jako atribut metody *alasql(stringWithSQL)* nebo jiné proměnné, pokud si je nazvete, třeba v Node.js. Dotazy lze vykonávat také nad jednotlivou tabulkou nějaké databáze pomocí funkce *exec()* (viz dále).

4.1.1 NodeJS

Knihovnu v NodeJS stačí pouze naimportovat, pomocí funkce *require()* o zbytek se vám postará správce balíčků. Použití může vypadat takto:

```
var alasql = require('alasql');
alasql('CREATE TABLE one (two INT)');
```

4.2 Nová databáze

Je tu možnost do proměnné založit novou databázi a nad vytvořenou proměnnou zavolat funkci *exec()* do které vložíme dotaz jako atribut jako v předchozí sekci (funkce *alasql()* je zkrácenou verzí *alasql.exec()*). Můžeme také vkládat více dotazů oddělených ";", kde poté navrácená hodnota bude jako pole výsledků a vy můžete mezi nimi iterovat.

```
var mybase = new alasql.Database();
mybase.exec('CREATE TABLE one (two INT)');
```

Také k nim můžete přistupovat přes globální objekt *alasql*. Obecně se všechny databáze nacházejí na cestě *alasql.databases.[název databáze]*. To k nim umožňuje jednodušší přístup přes objektovou cestu.

```
var mybase = new alasql.Database('mybase');
console.log(alasql.databases.mybase);
```

```
alasql: gi.Database {databaseid: "alasql", dbversion: 0, tables: {}, views: {}, triggers: {}, -}
db0: gi.Database {databaseid: "db0", dbversion: 0, tables: {}, views: {}, triggers: {}, -}
dbo: gi.Database {databaseid: "test2", dbversion: 0, tables: undefined, views: {}, triggers: {}, -}
myDatabase2: gi.Database {databaseid: "myDatabase2", dbversion: 0, tables: {}, views: {}, triggers: {}, -}
test: gi.Database {databaseid: "test", dbversion: 0, tables: undefined, views: {}, triggers: {}, -}
test2: gi.Database {databaseid: "test2", dbversion: 0, tables: undefined, views: {}, triggers: {}, -}
```

Obrázek 2: Inicializované databáze v objektu *alasql* (příklad)

4.3 Nová tabulka

Založení nové tabulky je podobné, můžeme ji vytvořit přímo v globálním objektu *alasql* nebo vytvořit uvnitř nějaké databáze. Budu předpokládat, že máme databázi uloženou v proměnné *var mybase*. Pozor, pokud tabulku založíte v globálním objektu *alasql*, lze k ní přistupovat a vykonávat dotazy přímo přes *alasql()*, pokud ale založíte tabulku v jiné databázi, musíte k ní přistupovat v

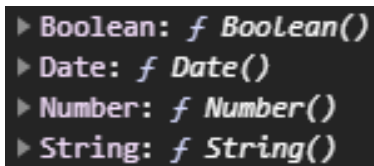
t= dané databázi (náš případ) pomocí funkce *exec()*. Založení tabulky je pak následující:

```
mybase.exec("CREATE TABLE cities (city string, population
number)");
console.log(alasql.databases.mybase.tables.cities);
console.log(mybase.exec("SELECT * FROM cities"));
```

Jak můžete vidět, výpis celé tabulky může být uskutečněn opět pomocí cesty *alasql.databases.[název databáze].tables.[název tabulky]* popř. *alasql.[název tabulky]* nebo jako SQL dotaz nad databází.

4.3.1 Datové typy

Datové typy, které jsou dostupné od začátku, tak se dají nalézt v *alasql* objektu pod cestou *alasql.fn.[název nového dat. typu]*. V základu jsou k dispozici tyto datové typy:



```
► Boolean: f Boolean()
► Date: f Date()
► Number: f Number()
► String: f String()
```

Obrázek 3: Základní datové typy

Příklad přidání nového datového typu do objektu databáze:

```
// nový datový typ
alasql.fn.Date = Date;
alasql('CREATE order (orderno INT, orderdate Date)');
```

4.4 Manipulace dat v tabulce

Jako jsme zvyklí z SQL, s daty jdou provádět různé úkony, jako je INSERT, UPDATE a DELETE. Syntaxe je stejná jako při založení nové tabulky. Akorát se změní dotaz. Jako příklad mohu uvést:

```
mybase.exec("INSERT INTO cities VALUES ('Rome',2863223),
('Paris',2249975), ('Berlin',3517424),
('Madrid',3041579)");
```

INSERT jako JS funkce

Knihovna umožňuje také vytvářet vlastní metody a jedna z možností je přes funkci *compile()* knihovny, která umožňuje překompilovat příkazy a přidá je do cache dané databáze.

```
var insert1 = db.compile('INSERT INTO one (?,?)');
var insert2 = db.compile('INSERT INTO one (:a,:b)');
```



```
insert1([1,2]);
insert2({a:3,b:4});
```

Funkce *compile()* nejspíše funguje, bohužel pokud jsem zavolaal funkci *insert()*, knihovna vyhodila chybu, a tvrdila, že to funkce není. Nenašel jsem nikde řešení.

Přidělení dat jako JS objekt

Určitě vás napadlo, že jako přistupujeme k databázím a tabulkám přes globální objekt, proč by nešlo přistupovat i k datům tabulky nebo je rovnou přiřadit a opravdu to jde. Obecně se všechna data tabulky nacházejí na cestě *alasql.databases.[název databáze].tables.[název tabulky].data*. Příklad objektu na přidělení dat může být následující:

```
const ceskaMesta = [
  {name: 'Prague', population: 1324277},
  {name: 'Brno', population: 381346},
  {name: 'Ostrava', population: 287968},
  {name: 'Plzen', population: 174842}
];
alasql.databases.mysql.tables.cities.data = ceskaMesta;
```

Je zde ale riziko, přidělená data neprochází přes žádnou funkci, tudíž data vůbec nemusí odpovídat schématu tabulky, při přidělení dat žádná výjimka nevyhodí, ale pokud by jsme nad daty zavolali nějaký dotaz, vyskočila by podmínka nebo by data byla *undefiend*. Také pokud vaše tabulka cities již nějaká data obsahuje, toto přiřazení daná data přemaže.

4.5 Načítání souborů

V dokumentaci je uvedené, že lze zřídit databázi z lokálního souborového systému nebo ze souboru SQLite (místo výroku FILESTORAGE -> SQLITE) a IndexedDB (místo výroku FILESTORAGE -> INDEXEDDB)

```
alasql('ATTACH FILESTORAGE DATABASE
test("./js/file.json")', function () {
  console.log(alasql.databases.test.data);
});
```

Vytvoří se databáze s názvem *test* s daty, která jsou v atributu data v dané databázi. Tento atribut pro jinak založené databáze není běžný, jelikož atribut data neobsahují, mají je až obsaženy v tabulkách databáze. Nepřišel jsem na způsob jak nad těmito daty použít dotazy, pouze jak přes adresu proměnné vypsat všechna data nebo dotazovat se pomocí placeholderu s odkazem na data. Je možné, že podporované databáze mají v souboru založené tabulky a Alasql je sám namapuje, pak lze nad nimi vyvolávat dotazy bez vyhození chyby.

5 Synchronní/Asynchronní přístup

Za normálního běhu knihovna vykonává procesy synchronně, ale jdou také volat callbacky nebo asynchronní volání pomocí metody *promise()*. Pokud pracujete se soubory, načítáte je, pak knihovna funguje asynchronně a je doporučeno používat již zmíněnou metodu *promise()*.

```
// callback
alasql('SELECT * FROM cities', [],
  function (res) {
    console.log(res);
  }
);

// promise
alasql.promise('SELECT * FROM cities')
  .then(function(res){
    // zpracovani response
  }).catch(function(err){
    // zpracovani chyby
  });
```

5.1 HTML

Knihovna dokáže číst data z HTML tabulky `<table>...</table>` a výsledek opět vygenerovat do HTML tabulky pomocí CSS selektoru.

```
// cteni z tabulky
alasql('SELECT * FROM HTML("#MyTable", {headers:true})');

// zapis do tabulky
alasql('SELECT * INTO HTML("#MyTable", {headers:true})
FROM ?', [data]);
```

6 SQL dotazy

Knihovna je založena, aby co nejvíce připomínala relační databáze s dotazovacím jazykem SQL, podporuje tedy všechny základní výroky SQL jazyka, jako jsou JOIN, GROUP, UNION, ANY, ALL, IN, zanožené dotazy a také správu transakcí. Dále také knihovna podporuje různé agregační metody a dimenzionální funkce, kterými jsou ROLLUP, CUBE a GROUPING SETS. Celý seznam podporovaných funkcí najdete na <https://github.com/agershun/alasql/wiki/SQL-99> SQL-99 features supported by AlaSQL.

6.1 Nejlépe funguje s JS

Tato knihovna je v podstatě SQL databáze v JavaScriptu. Slouží zejména jako podpora zpracování, usnadnění délky kódu, lepší práci s daty nebo jako samostatná client-side databáze - nejlépe však funguje **SPOLEČNĚ** s JavaScriptem. Knihovna nevrací rovnou tabulky, jak je běžné u relačních databází, ale

defaultně vrací data právě jako JSON a další práce s ním je už jako práce s objektem. Sami autoři ve své dokumentaci uvádějí nějaké příklady společně s JavaScriptem a pokud příliš neznáte knihovnu nebo nevíte že nějaké funkce obsahuje, je snazší zpracovat dotaz pomocí JavaScriptu.

```
// vypis poctu objektu ve vracenem poli
var db = new alasql.Database();
db.exec('select * from one', function(data) { // callback
    console.log(data.length);
});

// novy datovy typ
alasql.fn.Date = Date;
    alasql('CREATE order (orderno INT, orderdate Date)');
```

7 Datasetsy

7.1 Rohlík.cz

K dispozici máme data z webového portálu Rohlík.cz, který se zaměřuje na obchod s potravinami. Data obsahují dva soubory ze dne 2.11.2020, jeden z pohledu obyčejného uživatele (*rohlik2.json*) a druhý z pohledu prémiového uživatele (*rohlik1.json*), hlavní rozdíl mezi těmito dvěma datasety je v ceně za dopravu. Nejvíce nás zajímá atribut *availabilityDays*, který se sestává ze čtyř dalších objektů, popisující dnešek a další následující tři dny. V nich nalezneme atribut *slots*, ve kterém jsou objekty popisující hodiny daný den. V dané hodině nás zajímá atribut *timeSlotCapacityDTO*, který obsahuje konečný atribut *totalFreeCapacityPercent* popisující obsazenost danou hodinu. O úroveň výš je pak atribut *price* značící cenu dovozu.

7.2 Starcraft

Druhým datasetem jsou data ze hry Starcraft, máme k dispozici soubory dvou typů. Prvním typem (A) je popis jednotlivých **tierů**, kterých je dohromady 6, ale k dispozici máme pouze 5. a 6. Dále každý tier obsahuje pole po třech **divizích (úrovních)** kromě poslední 6. - ta obsahuje pouze jeden. Division má atributy *min_rating* a *max_rating* určující rozmezí hráčů. Každý division má pole obsahuje pole s několika **laddery**, které již obsahují jednotlivé hráče. Tyto laddery se v průběhu času mění a maximální počet hráčů v jednom ladderu je 100. Laddery popisují soubory typu B, které jsou ve dvou různých datech. Jednotlivé laddery se dají rozeznat podle id a timestampu. V souborech je uložené pole *team* které obsahuje data o jednotlivých hráčích, muhu uvést některé, které budu používat, např. *last_played_time_stamp* - udávající poslední aktivitu hráče v ms, *member[0].legacy_link.name* - nick hráče s rozeznávacím tagem, *member[0].played_race_count.race* - udávající rasu, za kterou hráč hrál, atd.

8 Důležitá je cesta, ne cíl

V této sekci popíši všechny dotazy, které jsem pomocí Alasql řešil, jak jsem postupoval a popř. co se mi nepovedlo nebo knihovna nenabízí.

8.1 Rohlík.cz

8.1.1 Jaké je nejbližší okno s volnou kapacitou?

Tento dotaz jsem musel řešit s pomocí JavaScriptu, jelikož struktura dat Rohlík.cz v *slots* (rozvozová okénka) jsou atributy a tudíž nelze procházet tento atribut v Alasql jako pole, ale pouze přímými adresami na jednotlivé objekty, pokud chceme projít všechny rozvozová okénka, musíme použít k iteraci JavaScript. V tomto dotazu nám jde o nalezení nejbližšího volného okénka, což je nalezení takové hodnoty, kdy *totalFreeCapacityPercent* je větší než 0:

```
(alasql('SELECT slots FROM ?'  
,[rohlikPremium.data.availabilityDays]).forEach(day => {  
  for (const [key, value] of Object.entries(day.slots)) {  
    const freeCapacity =  
      (alasql('SELECT timeSlotCapacityDTO->  
        totalFreeCapacityPercent  
        AS freeCapacity FROM ?',[value]));  
    if (freeCapacity[0].freeCapacity > 0) {  
      console.log('První volné okénko dne je: '  
        + value[0].since + ' do ' + value[0].till) ;  
      break;  
    }  
  }  
});
```

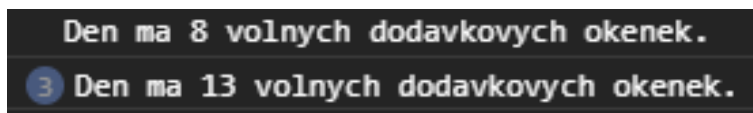
Prvním dotazem vybereme pomocí Alasql atribut *slots*, což je pole (dotaz je zde zbytečný, stačilo pouze přistoupit k poli pomocí JavaScriptu, ale chtěl jsem co nejvíce využívat Alasql). Procházíme každý den. Po té iterujeme každým časovým okénkem pomocí metody for pro iteraci objektů. V každém okénku si necháme pomocí dotazu napsaného v Alasql najít zanořený atribut *totalFreeCapacityPercent* a pomocí podmínky zjistit, jestli je větší jak 0, pokud ne opakujeme a pokud ano, tak necháme časové okénko vypsát a danou iteraci zrušíme pomocí break a iterujeme na další den. Tím se nám vypíší všechny okénka v každém dni:

```
První volné okénko dne je: 2020-11-02 13:00 do 2020-11-02 14:00  
První volné okénko dne je: 2020-11-03 08:00 do 2020-11-03 09:00  
První volné okénko dne je: 2020-11-04 08:00 do 2020-11-04 09:00  
První volné okénko dne je: 2020-11-05 08:00 do 2020-11-05 09:00
```

Obrázek 4: Výsledek dotazu Rohlík.cz - 1

8.1.2 Kolik volných časových intervalů je k dispozici?

Tento dotaz je podobný jako první, akorát nechceme pouze najít první volné časové okénko, ale všechny v ten daný den. Tudíž v podmínce nebudeme chtít iteraci přerušit, ale pouze aby pokračovala a pokud tam je volné okénko, tak ho přičíst k nějakému counteru a vypsát pro jednotlivý den, výsledek je:



```
Den ma 8 volnych dodavkovych okenek.  
3 Den ma 13 volnych dodavkovych okenek.
```

Obrázek 5: Výsledek dotazu Rohlik.cz - 2

8.1.3 Jaké je nejbližší nejlevnější okno s volnou kapacitou?

Tato otázka je podobná jako první, kdy chceme získat první volné okénko, ale k tomu ještě nejlevnější. K řešení musíme použít JSON, který není prémiový, jelikož prémiové členové mají dopravu zdarma. Postup je následující:

```
firstFreeDays = [];
(alasql('SELECT slots FROM ?', [rohlikNormal.data.availabilityDays])
  .forEach(day => {
    for (const [key, value] of Object.entries(day.slots)) {
      const freeCapacity = (alasql('SELECT timeSlotCapacityDTO ->
        totalFreeCapacityPercent AS freeCapacity FROM ?', [value
        ]));
      if (freeCapacity[0].freeCapacity > 0) {
        firstFreeDays.push({od: value[0].since,
          do: value[0].till,
          price: value[0].price});
        break;
      }
    }
  }));
lowPriceFree = alasql('SELECT * FROM ? WHERE price=(SELECT MIN(
  price) FROM ?)', [firstFreeDays, firstFreeDays]);
console.log('První nejlevnější okno je: od ' + lowPriceFree[0].od +
  ' do ' + lowPriceFree[0].do + ' s cenou ' + lowPriceFree[0].
  price + ' Kc.');
```

Nejdříve si založíme pole, kam budeme dávat volná okénka pro každý den. Postup je zde stejný jako u první otázky, akorát místo toho aby jsme okénko jen vypsalí, uložíme si ho do pole. Po iteraci nad polem zavoláme dotaz, který nám vybere objekt s nejmenší cenou a ten si vypíšeme. Výsledek je takový:

```
První nejlevnější okno je: od 2020-11-04 08:00 do 2020-11-04 09:00 s cenou 39 Kc.
```

Obrázek 6: Výsledek dotazu Rohlik.cz - 3

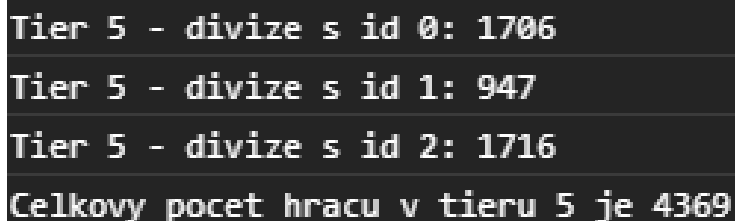
8.2 Starcraft

8.2.1 Kolik hráčů je v jednotlivých divizích/úrovních?

V této otázce chceme najít celkový počet hráčů na úrovni a v jejích divizích. Nejdříve vybereme všechny divize dané úrovně, poté v každé divizi sečteme počet hráčů ve všech ladderech a nakonec obojí vypíšeme. Vstupním parametrem je tier.

```
function sumPlayersFromDivision(tier) {  
  let counter = 0;  
  for (let i = 0; i < tier.tier.length; i++) {  
    memberCount = alasql('SELECT SUM(member_count) AS  
      member_count FROM ?', [tier.tier[i].division]);  
    console.log('Tier ' + tier.key.league_id + ' - divize s id  
      ' + tier.tier[i].id + ': ' + memberCount[0].  
      member_count);  
    counter += memberCount[0].member_count;  
  }  
  console.log('Celkový počet hrácu v tieru ' + tier.key.league_id  
    + ' je ' + counter);  
}
```

Zřídíme si counter pro počet hráčů na úrovni. Dále si for cyklem spočítáme/vybereme počet hráčů pomocí funkce SUM() v každé divizi, index zde slouží pro iteraci mezi divizemi. Vypíšeme počet hráčů v divizi a přičteme do counteru. Za tělem for cyklu vypíšeme count, který značí všechny hráče v úrovni. Výsledek je takový:



```
Tier 5 - divize s id 0: 1706  
Tier 5 - divize s id 1: 947  
Tier 5 - divize s id 2: 1716  
Celkový počet hrácu v tieru 5 je 4369
```

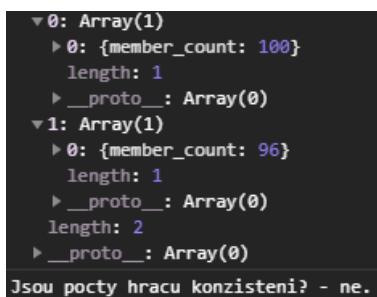
Obrázek 7: Výsledek dotazu Starcraft - 1

8.2.2 Jsou konzistentní počty hráčů mezi JSON typ A a B?

V otázce nás zajímá, zda počet `members_count` v JSONu typu A je stejný jako skutečný počet hráčů v poli `team` v JSON typu B. V AlaSQL lze řešit tuto otázku pomocí jednoho volání s dvěma dotazy.

```
function ladderMemberCountConsist(tier, ladderId) {
  count = alasql('SELECT member_count FROM ? WHERE ladder_id= ?;
    SELECT COUNT(*) AS member_count FROM ?', [tier.tier[0].
    division, ladderId, ladder230898_1031.team])
  console.log(count);
  console.log('Jsou pocky hracu konzistenti? - ' + (count[0][0].
    member_count === count[1][0].member_count ? 'ano.' : 'ne.'
  ));
}
```

Určím si id ladderu, který mě zajímá a ten naleznu v JSONu typu A pomocí klauzule `WHERE` a vyberu `member_count`. Je zde ale problém, že musím přímo zadat index `division`, ve kterém se ladder nachází. Tento problém blíže popisují v kapitole [Chyby](#), jelikož knihovna neumí 2x iterovat. Také se musí přidat ladder (JSON typu B). V druhém dotazu si spočtu pomocí `COUNT()` počet záznamů v poli `team` a vypíši. Pro výpis ještě použiji podmínku `if`, která výsledek vypíše slovně. Ladder zde musí být vložen manuálně, jelikož nepracuji se souborovým systémem, řešením by bylo procházet složku s daty a hledat soubor s id daného ladderu. Výsledkem ladderu s id 230898 je:



```
▼ 0: Array(1)
  ► 0: {member_count: 100}
    length: 1
  ► __proto__: Array(0)
▼ 1: Array(1)
  ► 0: {member_count: 96}
    length: 1
  ► __proto__: Array(0)
length: 2
► __proto__: Array(0)
Jsou pocky hracu konzistenti? - ne.
```

Obrázek 8: Výsledek dotazu Starcraft - 2

8.2.3 Seřazení hráčů v rámci jednoho ladderu.

S touto otázkou si knihovna poradí celkem snadno (bez použití JavaScriptu), pokud chceme seřadit hráče v jednom ladderu a známe id tohoto ladderu. S příloženými daty si například vezmeme soubor *ladders-eu-230898.json*, což je ladder s *ladder_id=230898* a pomocí JS můžeme přistupovat k jeho atributům. Všechny hráči daného ladderu jsou uloženy v poli *team*. Chceme zobrazit id hráče, k případnému dalšímu dotazování, jméno hráče a rating, což bude seřazená hodnota. Vytvoříme si AlaSQL dotaz:

```
function orderPlayersFromLadder(ladderJson) {  
  console.log(alasql('SELECT id, rating, member->(0)->  
    character_link->battle_tag AS battle_tag FROM ? ORDER BY  
    rating DESC', [ladderJson.team]));  
}
```

Vybereme všechny zmíněné atributy. Atribut jména hráče je zanořený, proto musíme volit přístup přes další atributy nebo pole a zobrazíme ho jako *battle_tag*, jinak by se celý atribut nazýval podle cesty, které k němu přistupujeme (v tomto případě by se vybraný atribut jmenoval 'member->(0)->character_link->battle_tag'). Vybraná data můžeme seřadit podle atributu *rating*, pomocí výrazu *ORDER BY* sestupně nebo vzestupně. Jako výsledek dotazu nad ladderem s *ladder_id=230898* nám dotaz sestojí následující výsledek: Jediné na co

```
▶ 0: {id: "15100607019867963392", rating: 6015, battle_tag: "Morris#21456"}  
▶ 1: {id: 6131688361959621000, rating: 5881, battle_tag: "1111111111769"}  
▶ 2: {id: "17590824934920159232", rating: 5867, battle_tag: "hodor#2879"}  
▶ 3: {id: "16861226402123350016", rating: 5800, battle_tag: "WingsOffFire#21508"}  
▶ 4: {id: 8640186757335220000, rating: 5771, battle_tag: "Jun0#1480"}  
▶ 5: {id: 6695757618218009000, rating: 5641, battle_tag: "Tartem#2108"}  
▶ 6: {id: 5761260695538631000, rating: 5620, battle_tag: "JustLeaveNow#2171"}  
▶ 7: {id: 3229681352072757000, rating: 5592, battle_tag: "Progamer#22909"}  
▶ 8: {id: "12505976182107275264", rating: 5560, battle_tag: "froz#2369"}  
▶ 9: {id: 8720978871744201000, rating: 5558, battle_tag: "endrju#2412"}  
▶ 10: {id: 2602275828007960600, rating: 5550, battle_tag: "Irbis#1136"}  
▶ 11: {id: 1010527138127806500, rating: 5547, battle_tag: "Tranzistor#21474"}  
▶ 12: {id: "15357872949558247424", rating: 5481, battle_tag: "111111111111#21429"}  
▶ 13: {id: 5004659256675271000, rating: 5414, battle_tag: "Ditrih#21911"}  
▶ 14: {id: 4827608197789385000, rating: 5378, battle_tag: "digbickdaddy#2501"}  
▶ 15: {id: "13837623301809111040", rating: 5370, battle_tag: "Mookashade#2499"}  
▶ 16: {id: "15953199122371051520", rating: 5355, battle_tag: "Surprise#2673"}  
▶ 17: {id: 7345120389489492000, rating: 5355, battle_tag: "Mantaza#2752"}  
▶ 18: {id: "10226861201053188096", rating: 5337, battle_tag: "Shoes#2851"}  
▶ 19: {id: "11166708347313324032", rating: 5331, battle_tag: "Goomba#1920"}  
▶ 20: {id: 4725715355731755000, rating: 5317, battle_tag: "Protoss#21530"}  
▶ 21: {id: 8569255063204135000, rating: 5310, battle_tag: "NEEK#2928"}  
▶ 22: {id: "10303985344671907840", rating: 5305, battle_tag: "Seracis#2371"}  
▶ 23: {id: "13903487346847776768", rating: 5294, battle_tag: "JustLeaveNow#2171"}  
▶ 24: {id: 4411589281722663000, rating: 5251, battle_tag: "Project#21119"}  
▶ 25: {id: 8641312657242063000, rating: 5250, battle_tag: "Bloop#2209"}  
▶ 26: {id: "15722103768445091840", rating: 5238, battle_tag: "RhCn#2502"}  
▶ 27: {id: "16455626458241433600", rating: 5230, battle_tag: "HänTattoo49#2314"}  
▶ 28: {id: 5441506221506953000, rating: 5204, battle_tag: "Dext#2339"}  
▶ 29: {id: "11453247674581516288", rating: 5204, battle_tag: "DasDuelon#2923"}  
▶ 30: {id: 8205594890852106000, rating: 5168, battle_tag: "Fortuna#2755"}
```

Obrázek 9: Výsledek dotazu Starcraft - 3

jsem během tohoto dotazu narazil je, že jsem chtěl, aby si uživatel volil metodu seřazení, jestli ASC nebo DESC přes placeholder pomocí ?, bohužel ale při překladi aplikace padala a vyžadovala již definovanou metodu seřazení vloženou přímo jako string v dotazu.

8.2.4 Kolik hráčů odehrálo v poslední době nějakou hru?

K tomuto dotazu knihovně opět postačí pouze jeden dotaz, se dvěma vstupními parametry, které nahradíme místo placeholderů a to určitý ladder a konstanta času v ms proti které budeme hráče porovnávat. Alasql dotaz je:

```
function lastPlayedFrom(ladderJson, timeFromInMilis) {  
  console.log(alasql.exec('SELECT COUNT(*) FROM ? WHERE  
    last_played_time_stamp > ?', [ladderJson.team,  
    timeFromInMilis]));  
}
```

Chceme zobrazit číslo, tedy počet hráčů, kolik hrálo Starcraft od časové značky. Použijeme agregační funkci COUNT buď s nějakým atributem nebo hvězdičkou, což je v tomto případě jedno. Musíme přidat také klauzuli WHERE a říci, že hledáme hráče, kteří mají *last_played_time_stamp* větší než naše časové razítko. Výsledkem dotazu je:

A screenshot of a console log output. It shows a single line of text: ▶ 0: {COUNT(*): 16}. The text is in a monospaced font, with the opening curly brace and the number 16 highlighted in blue.

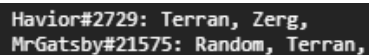
Obrázek 10: Výsledek dotazu Starcraft - 5

8.2.5 Jací hráči jsou v jednom ladderu vícekrát a za jaké rasy?

Tento dotaz jsem se snažil vyřešit pouze s pomocí Alasql, ale k výsledku jsem se dobral pouze s pomocí JavaScriptu. Vstupním parametrem je jeden ladder.

```
function playerMultiRace(ladderJson) {  
    result = '';  
    temp = alasql('SELECT member->(0)->character_link->battle_tag  
as name, member->(0)->played_race_count->(0)->race as race  
FROM ?', [ladderJson.team]);  
    alasql('SELECT name FROM ? GROUP BY name HAVING COUNT(*) > 1  
ORDER BY name', [temp]).forEach(dupPlayer => {  
        result += dupPlayer.name + ': '  
        temp.forEach(player => {  
            if (dupPlayer.name === player.name) {  
                result += player.race + ', '  
            }  
        });  
        result += '\n';  
    });  
    console.log(result);  
}
```

Nejdříve inicializuji prázdný string, který mi bude pomáhat s vypsáním výsledků dotazů. Nejdříve si do proměnné temp zobrazím *battle_tag* hráčů jako *name* a rasu jako *race* za kterou hráči hráli. V dalším dotazu zobrazím *name* hráče, který se vyskytl v daném ladderu vícekrát. Po té si pomocí JavaScriptu pro každého hráče nechám vypsát jméno a rasy, za které hrál. Výsledek nad ladderem s ladder_id=230882 vypadá takto:



```
Havior#2729: Terran, Zerg,  
MrGatsby#21575: Random, Terran,
```

Obrázek 11: Výsledek dotazu Starcraft - 6

8.2.6 Jací hráči přibyli/ubyli v daných ladderech?

V tomto dotazu chceme porovnat dvě instance stejného ladderu v jiných časových obdobích, tudíž vstupem budou dva zmíněné parametry. Tímto dotazem si Alasql poradí pomocí dvou dotazů nebo jedním zanořeným:

```
function compareLaddersByPlayers(ladderJson1, ladderJson2) {  
  union = (alasql('SELECT member->(0)->character_link->  
    battle_tag as name FROM ? UNION ALL SELECT member->(0)->  
    character_link->battle_tag as name FROM ?', [ladderJson1.  
    team, ladderJson2.team]));  
  console.log(alasql('SELECT name FROM ? GROUP BY name HAVING  
    COUNT(*)=1', [union]));  
}
```

Prvním krokem se sestavení UNION výroku, který vezme z obou instancí battle_tag hráčů jako *name* a vytvoří z nich pouze jednu tabulku. Nad touto tabulkou zobrazíme pouze ty hráče, kteří se v obou instancích nacházejí pouze jednou. Nevíme ale, jestli daní hráči přibili nebo ubyli. Výsledek dotazů mezi ladderem s ladder_id=230882 je:

```
▶ 0: {name: "Intebasnygg#2196"}  
▶ 1: {name: "siao#2503"}  
▶ 2: {name: "LuPin#2455"}
```

Obrázek 12: Výsledek dotazu Starcraft - 7

8.2.7 Jací hráči vybočují z hranic divizí?

V dotazu chceme nalézt všechny hráče, kteří patří do jedné divize. Toho docílíme nalezením všech ladderů dané divize. Také potřebujeme získat ELO rozsah dané divize pomocí ukazatelů `min_rating` a `max_rating`. Až budeme mít všechny hráče z ladderů dané divize, lze jednoduše podle jejich ratingu vyfiltrovat hráče, kteří do rozsahu divize nezapadají.

```
function playersElo(tier, divisionId) {
  division = alasql('SELECT * FROM ? WHERE id= ?', [tier.tier,
    divisionId]);
  divisionLaddersId = alasql("SELECT ladder_id AS lad_id FROM ?",
    [division[0].division]);
  // nebo - vybereme divizi a JOINem spojíme s ladder ID (SELECT
  // ladder.team FROM ladder INNER JOIN division ON division.
  // ladder=ladder.id)
  // divisionLadders.forEach(ladder => {
  console.log(alasql('SELECT member->(0)->legacy_link->name AS
    name, rating FROM ? WHERE rating NOT BETWEEN ? AND ?', [
      ladder230882_1101.team, division[0].min_rating, division
      [0].max_rating]));
  // });
}
```

Naleznu divizi nebo divize určitého tieru a vyberu min/max každé úrovně, také pole `division` se všemi laddery kvůli jejich id. Dalším krokem je nalezení všech ladderů z divize.

V semestrální nepracuji se souborovým systémem a musel bych každý ladder zadat manuálně. Tento krok by šel udělat např. vybrat všechny soubory na základě `ladder_id` a načíst je pro další zpracování nebo z jednoho velkého souboru ladderů vybrat ty, které mají stejné id (což by bylo asi časově náročnější). Bohužel v takovém případě nelze udělat JOIN (více tento problém popisují v kapitole [Chyby](#)), důvodem je, že pracujeme nad JSON souborem a ne nad tabulkou - nemáme název tabulky.

Nakonec pro každý ladder vypíšeme name a rating hráčů, kteří nejsou v rozsahu `min_rating` a `max_rating` divize. Toho docílíme pomocí NOT BETWEEN operátoru. Dané hráče vypíšeme. Výsledek pro jeden ladder je:

```

> 0: {name: "Intebarasnyg#281", rating: 4322}
> 1: {name: "Eladen#347", rating: 4356}
> 2: {name: "NinjaSpectre#946", rating: 4306}
> 3: {name: "ugly#860", rating: 4297}
> 4: {name: "Vulcan#128", rating: 4234}
> 5: {name: "Floppgun#498", rating: 4149}
> 6: {name: "NewbRevenge#100", rating: 4163}
> 7: {name: "TAS#850", rating: 4157}
> 8: {name: "Warembur#817", rating: 4114}
> 9: {name: "Angelus#1292", rating: 4334}
> 10: {name: "IIIIIIIIII#1660", rating: 4335}
> 11: {name: "EseEseEse#738", rating: 4328}
> 12: {name: "CheekClapper#513", rating: 4312}
> 13: {name: "Pyrom#992", rating: 4255}
> 14: {name: "CROWOS#1889", rating: 4306}
> 15: {name: "Aquaafresh#519", rating: 4318}
> 16: {name: "Swing#961", rating: 4318}
> 17: {name: "schoy#899", rating: 4198}
> 18: {name: "Mrkao#960", rating: 4253}
> 19: {name: "Paddington#567", rating: 4352}
> 20: {name: "Amator#459", rating: 4236}
> 21: {name: "ACAZ#781", rating: 5037}
> 22: {name: "Briik#476", rating: 4354}

```

Obrázek 13: Výsledek dotazu Starcraft - 4

8.2.8 Kolik zástupců mají týmy v jednotlivých úrovních?

Tento dotaz je postupem velice podobný tomu předešlému, musíme nalézt všechny teamy, které jsou v dané diviti a spočítat počet jejich členů. Prvním krokem je výběr divize a nalezení všech ladderů daných divizí v úrovni. Poté získáme všechny clan_tagy (předpokládám, že funguje jako jednoznačný identifikátor). Po získání všech clan_tagu necháme je sloučit a sečíst, kolikrát se vyskytují, tím získáme počet hráčů. Jako vstupní parametr nám tedy stačí úroveň.

```

function teamsInTier(tierJson) {
  // vypis vseh ladderu
  let tierLadders = [];
  tierJson.tier.forEach(division => {
    tierLadders.push(alasql("SELECT ladder_id AS lad_id FROM ?"
      , [division.division]));
  });
  // JOIN pres tabulku, kde prvni bude ladder id a druhej sloupec
  // bude nazev ladder souboru
  // SELECT tag, id, COUNT(*) FROM tier INNER JOIN ladders ON
  // league_id=league_id GROUP BY tag
  let teamsMembersCount = [];
  console.log(alasql('SELECT member->(0)->clan_link->clan_tag AS
    tag, member->(0)->clan_link->id AS id ,COUNT(*) AS members
    FROM ? GROUP BY member->(0)->clan_link->clan_tag',[
      ladder230889_1031.team]));
  alasql('SELECT member->(0)->clan_link->clan_tag AS tag, member
    ->(0)->clan_link->id AS id ,COUNT(*) AS members FROM ?
    GROUP BY member->(0)->clan_link->clan_tag',[
      ladder230889_1031.team]).forEach(item => {
    teamsMembersCount.push(item);
  });
  alasql('SELECT member->(0)->clan_link->clan_tag AS tag, member
    ->(0)->clan_link->id AS id ,COUNT(*) AS members FROM ?
    GROUP BY member->(0)->clan_link->clan_tag',[

```

```

        ladder230898_1031.team]).forEach(item => {
            teamsMembersCount.push(item);
        });
        alasql('SELECT member->(0)->clan_link->clan_tag AS tag, member
->(0)->clan_link->id AS id ,COUNT(*) AS members FROM ?
GROUP BY member->(0)->clan_link->clan_tag',[
        ladder230882_1031.team]).forEach(item => {
            teamsMembersCount.push(item);
        });
        // nad polem se zgrupujou týmy a secte vyskyt hracu
        console.log(alasql('SELECT tag, SUM(members) FROM ? GROUP BY
tag', [teamsMembersCount]));
    }
}

```

V prvním kroku vybereme všechny divize úrovně a k nim dané laddery. Tento krok je stejný jako u předchozí otázky, akorát pro všechny divize dané úrovně. V tomto kroku je také problém jako u minulé otázky, jak vybrat všechny laddery na základě id? Problém a postupy jsem popisoval u předchozí otázky, tudíž to sem nebudu psát znovu. Nad každým ladderem vybereme clan_tag (které jsou v JSON souborech typu B pod každým hráčem) a počet hráčů pomocí metody COUNT(). Navíc s využitím GROUP BY si vybereme všechny týmy jen jednou. Po té co máme takto zpracované všechny laddery dané úrovně, můžeme je grupovat podle jejich tagu a pomocí agregační metody SUM sečíst výskyt hráčů. V datech se vyskytují hráči, kteří žádný tým nemají, v zadání otázky se na to netážeme, ale pro informaci se může hodit znát. Jako výstup ukáží pouze část výsledků ze 3 ladderů:

```

> 0: {tag: undefined, SUM(members): 151}
> 1: {tag: "KΠCC", SUM(members): 1}
> 2: {tag: "cSc!", SUM(members): 3}
> 3: {tag: "HBGSJR", SUM(members): 1}
> 4: {tag: "Based", SUM(members): 1}
> 5: {tag: "UATeam", SUM(members): 1}
> 6: {tag: "MkersA", SUM(members): 1}
> 7: {tag: "FTW", SUM(members): 1}
> 8: {tag: "TheDOH", SUM(members): 1}
> 9: {tag: "Wolven", SUM(members): 3}
> 10: {tag: "RöckeT", SUM(members): 1}
> 11: {tag: "aW", SUM(members): 4}
> 12: {tag: "TuS", SUM(members): 1}
> 13: {tag: "Hasu", SUM(members): 4}
> 14: {tag: "Pw", SUM(members): 1}
> 15: {tag: "ICG1", SUM(members): 2}
> 16: {tag: "Heroes", SUM(members): 2}

```

Obrázek 14: Výsledek dotazu Starcraft - 8

8.3 Objevené zajímavé funkce

Placeholder

Otazník v dotazech je tzv. placeholder. Všechny dotazy, které jsou volány přes Alasql mohou mít v dotazu otazník, který je pak nahrazen hodnotou v poli za výrazem, které obsahuje odkazy na zdroje.

```
alasql('SELECT a->b FROM ?', [data]);
```

Vlastní funkce

AlaSQL je rozšiřitelná a umožňuje nám vytvářet vlastní funkce s použitím JavaScriptu, které pak můžeme zavolat přímo v SQL dotazu. Tato funkcionality umožňuje široké škálování knihovny a pokud máte s JavaScriptem zacházet, umožňuje vám to vyřešit skoro jakýkoliv problém.

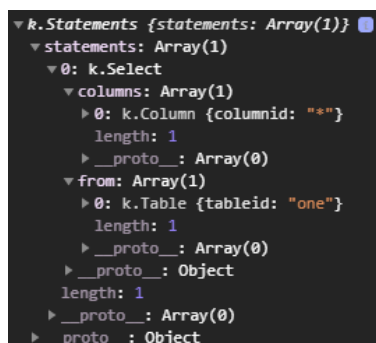
```
alasql.aggr.CONCAT = function(v,s) {  
    return (s||[]).concat(v);  
};  
  
alasql.aggr.LENGTH = function(array) {  
    return array.length;  
}
```

Abstract syntax tree

Umožňuje převádět jazyk SQL na AST (Abstract Syntax Tree), který můžeme programově interpretovat do našeho datového modelu.

```
var ast = alasql.parse("SELECT * FROM one");  
console.log(ast.toString()); // Vypise puvodni SQL dotaz
```

Tato funkce se může použít různě, například pro rychlejší sestavování větších dotazů. Pro představu uvedu strukturu rozloženého objektu:



Obrázek 15: Struktura AST stromu v JavaScriptu

pretty(sql)

Interně, pokud se ve svých dotazech vyznáme, je tato funkce asi zbytečná, ale je dobrá ji mít při vypisování, speciálně dlouhých, dotazů do konzole nebo kamkoliv jinam. Funkce totiž zkráší náš stringový dotaz.

Místo funkcí *alasql()* nebo *exec()* můžete využít také jednu z následujících funkcí:

```
alasql.value(sql, params, callback)
alasql.log(sql, params)
```

První funkce *value()* vykoná dotaz, ale místo objektu nebo více hodnot vrátí pouze jednu. Druhá funkce *log()* opět vykoná dotaz a rovnou výsledek vypíše do konzole nebo přímo do HTML elementu. Což se velice hodí při debugování.

9 Chyby

9.1 Procházení zanořených objektů

Jako největší chybou nebo nedostatkem jsem se potýkal s problémem, že knihovna nedokáže procházet přes zanořený JSON pomocí jednoho dotazu. Představme si, že máme pole měst a v každém městě je ještě pole okrsků, pokud by jsme chtěli zjistit/vypsát např. všechny názvy okrsků, šli by jsme na to v AlaSQL nějak takto:

```
alasql('SELECT okrsky->name FROM (SELECT okrsky FROM ? WHERE name
      LIKE 'Praha'), [mesta]);
```

Předpokládá se, že knihovna nejdříve vybere všechny okrsky ve městě a pak pro každý okrsek vypíše jméno, bohužel takto jména okrsků nelze získat, pouze to jde nad jedním prvkem v poli, kdy by jsme si museli určit přímo index, který chceme:

```
alasql('SELECT okrsky->[0]->name FROM (SELECT okrsky FROM ? WHERE
      name LIKE 'Praha'), [mesta]);
```

Takto dotaz již vrátí výsledky, ale pouze pro první okrsek daného města. V dokumentaci jsem bohužel nenašel žádný způsob, jak by se např. dalo říci, že chci projít celé pole... našel jsem pouze metodu *arrayOfArray*, ale ta bohužel nefungovala (viz. další kapitola). Proto musíme místo toho použít nějakou iterační metodu JavaScriptu, např. *forEach*. Vybrali by jsme tedy všechny okrsky daného města a nad tímto výsledkem proiterovali dotaz, který by nám vypsal jednotlivé názvy měst.

```
alasql('SELECT okrsky FROM ? WHERE name LIKE 'Praha'), [mesta]).
  forEach(okrsek => {
    console.log(alasql('SELECT name FROM ?, [okrsek]);
  });
```

Tento problém, ale právě řeší JavaScript, který lze bez problému použít, proto možná toto knihovna zbytečně neimplementuje.

9.2 Nefunkční

Další nesnází je nefunkčnost některých metod, které tvůrci uvádějí v dokumentaci, ale pokud je chcete použít, knihovna vyhodí chybu. Mezi takovéto funkce patří například již zmiňovaná *arraayOfArray()*. Funkce nešli, ani když jsem měl nejnovější verzi knihovny.

9.3 JOIN

Funkce JOIN zde existuje a funguje, tudíž bych nechtěl říkat, že je to chyba, ale funguje pouze s vytvořenými tabulkami které máte pojmenované, JSON soubory, které zadáváte přes placeholder bohužel nelze joinovat, jelikož postrádají název. Jako příklad použití JOINu mohu uvést:

```
alasql('SELECT city.*, country.* FROM city JOIN country USING  
countryid');  
alasql('SELECT * FROM Cities JOIN Countries USING Country');
```

10 Popis přiložených souborů

K této dokumentaci také přikládám soubory s příklady a s řešením popsanych otázek. Struktura zip souboru je následující. Po rozbalení je zde jedna složka ..., která je jako root složka projektu v ní se nacházejí složky *dokumentace* - obsahuje dokumentaci ve formátu .pdf, *main* - obsahující HTML a JavaScript soubory a složku *data* - obsahující data poskytnutá k této semestrální práci. pokud půjdeme do složky *main* najdeme zde složky *data* - která obsahuje některá z poskytnutých dat převedená do JavaScriptu, *js* - obsahující hlavní soubory s kódem a řešení. V main složce jsou také tři HTML soubory, které po otevření v některém z prohlížečů a otevření záložky Console v DevToolu prohlížeče najdete vypsané všechny výsledky dotazů.

11 Porovnání s SQL

Knihovna je primárně navržena aby se s ní co nejvíce pracovalo jako s SQL. Obsahuje všechny základní výroky (SELECT, DISTINCT, FIRST, FROM, WHERE, ORDER BY, atd.) agregační funkce (SUM(), AVG(), GROUP BY(), atd.), stringové, numerické, logické a datové funkce (ABS(), UPPER(), YEAR(), ...), JOIN operace (INNER JOIN, LEFT JOIN, OUTER JOIN, ...) a řízení transakcí příkazy (BEGIN, COMMIT a ROLLBACK).

Jak jsem již zmínil v úvodu, knihovna implementuje mnoho funkcí ze čtvrté verze jazyka SQL (SQL:1999). Přehled, co všechno knihovna podporuje, lze nalézt [zde](#).

Pokud si vytvoříte databázi pomocí Alasql, chová se stejně jako SQL databáze, akorát ukládání dat je jiné. Data jsou uloženy v JavaScriptovém objektu a máme k nim přístup i bez dotazovacího jazyka přes objektovou cestu. Data je možné ukládat i do Localstorage nebo jiné podporované databáze.

Jak již bylo také zmíněno, knihovna vrací výsledky dotazů jako objekt nebo pole výsledků.

12 Na závěr

Celkově se mi s knihovnou dělalo relativně dobře, ALE... pouze při práci společně s JavaScriptem. Dokumentace knihovny pokrývá základy a místy je vidět, že autoři mají připravené odkazy a stránky pro její rozšíření, bohužel je v ní takový celkem zmatek, některé věci jsou na více místech a většina z "live example" nefungují. Odkaz na knihovnu z wiki je ve verzi 0.3.9, což je starší verze, přitom píš, že verze v npm je verzi 0.6.4. Jediné relevantní stránky jsou ty na GitHubu, ostatní stránky v silné většině kopírují obsah a nelze načíst další relevantní zdroj. Bohužel, což mě hodně zklamalo a trochu odradilo jsou funkce, které uvádějí se zajímavými, ne-li potřebnými funkcemi a bohužel, když si je naimplementujete do svého kódu, tak vám to většinou vyhodí exception, že požadovaná funkce zavolaná nad objektem *alasql* není funkcí a nefunguje.

Na druhou stranu, knihovna nabízí opravdu mnoho zajímavých funkcí a pomocí JavaScriptu jde udělat prakticky cokoliv. Je zde vždy několik cest, jak s knihovnou pracovat, to znamená, že je možné různé problémy řešit různými cestami. Knihovna je velmi populární, podle statistik si ji stáhne cca 13 000 lidí týdně. Leč na knihovně se stále pracuje (soudě podle commitů) a věřím, že autoři opraví stávající problémy a přidají ještě další funkce. Knihovna je open source, takže kdyby jste potřebovali nějaký projekt na ní založit, můžete. Také pokud vám tam něco chybí, není těžké si udělat vlastní funkci, která toto vykoná. Knihovna se z velké části používá jako normální SQL dotazy a uživatel JavaScriptu s ní nebude mít také problémy.

Když jsem vybíral tuto knihovnu, knihovna mě velice zajímala a myslel jsem si, že práce s JSONem nebude problém, bohužel jak jsem začal na semestrální práci pracovat, zjišťoval jsem, že knihovna je spíše založená pro správu relační databáze a úschovu vlastních dat do *LocalStorage* prohlížeče a transformace dat z tabulkových (xls, csv) souborů. Není to přímo určena pro zpracování JSONů, ale základní zpracování umožňuje.

V poslední řadě bych chtěl zmínit špatné chybové hlášky, kdy jsem nevěděl co takový hláška říká. Mnohdy se vám vypíše jen písmenka a pokud nemáte nastudovanou přímo scripty knihovny, tak nevíte, proč to padá. Jako příklad uvádím:

```
Uncaught ReferenceError: g is not defined
    at gi.Query.eval [as wherefn] (eval at k.Select.compileWhere (alasql.min.js:2), <anonymous>:3:7)
    at r (alasql.min.js:2)
    at r (alasql.min.js:2)
    at D (alasql.min.js:2)
    at I (alasql.min.js:2)
    at Object.eval [as datafn] (eval at <anonymous> (alasql.min.js:2), <anonymous>:3:57)
    at alasql.min.js:2
    at Array.forEach (<anonymous>)
    at alasql.min.js:2
    at a (alasql.min.js:2)
```

Obrázek 16: Příklad vyhozené chybové hlášky

Reference

- [1] *Github - AlaSQL* [online]. [cit. 2020-11-16]. Dostupné z: <https://github.com/agershun/alasql>
- [2] *Github - AlaSQL/Wiki* [online]. [cit. 2020-11-16]. Dostupné z: <https://github.com/agershun/alasql/wiki>
- [3] *W3C - SQL* [online]. [cit. 2020-11-16]. Dostupné z: <https://www.w3schools.com/sql/default.asp>
- [4] *cdnjs - alasql* [online]. [cit. 2020-11-16]. Dostupné z: <https://cdnjs.com/libraries/alasql>
- [5] *DZone - alasql-in-action-the-javascript-sql-database* [online]. [cit. 2020-11-16]. Dostupné z: <https://dzone.com/articles/alasql-in-action-the-javascript-sql-database>
- [6] *DEV - alasql-a-real-database-for-web-browsers-and-nodejs* [online]. [cit. 2020-11-16]. Dostupné z: https://dev.to/jorge_rockr/alasql-a-real-database-for-web-browsers-and-nodejs-24gj

Seznam obrázků

1	Logo AlaSQL	4
2	Inicializované databáze v objektu <i>alasql</i> (příklad)	6
3	Základní datové typy	7
4	Výsledek dotazu Rohlik.cz - 1	11
5	Výsledek dotazu Rohlik.cz - 2	12
6	Výsledek dotazu Rohlik.cz - 3	13
7	Výsledek dotazu Starcraft - 1	14
8	Výsledek dotazu Starcraft - 2	15
9	Výsledek dotazu Starcraft - 3	16
10	Výsledek dotazu Starcraft - 5	18
11	Výsledek dotazu Starcraft - 6	19
12	Výsledek dotazu Starcraft - 4	20
13	Výsledek dotazu Starcraft - 7	21
14	Výsledek dotazu Starcraft - 8	23
15	Struktura AST stromu v JavaScriptu	24
16	Příklad vyhozené chybové hlášky	27