Graal v2 : Chase and storage

Developments and perspectives

Florent TORNIL

September 21 2021

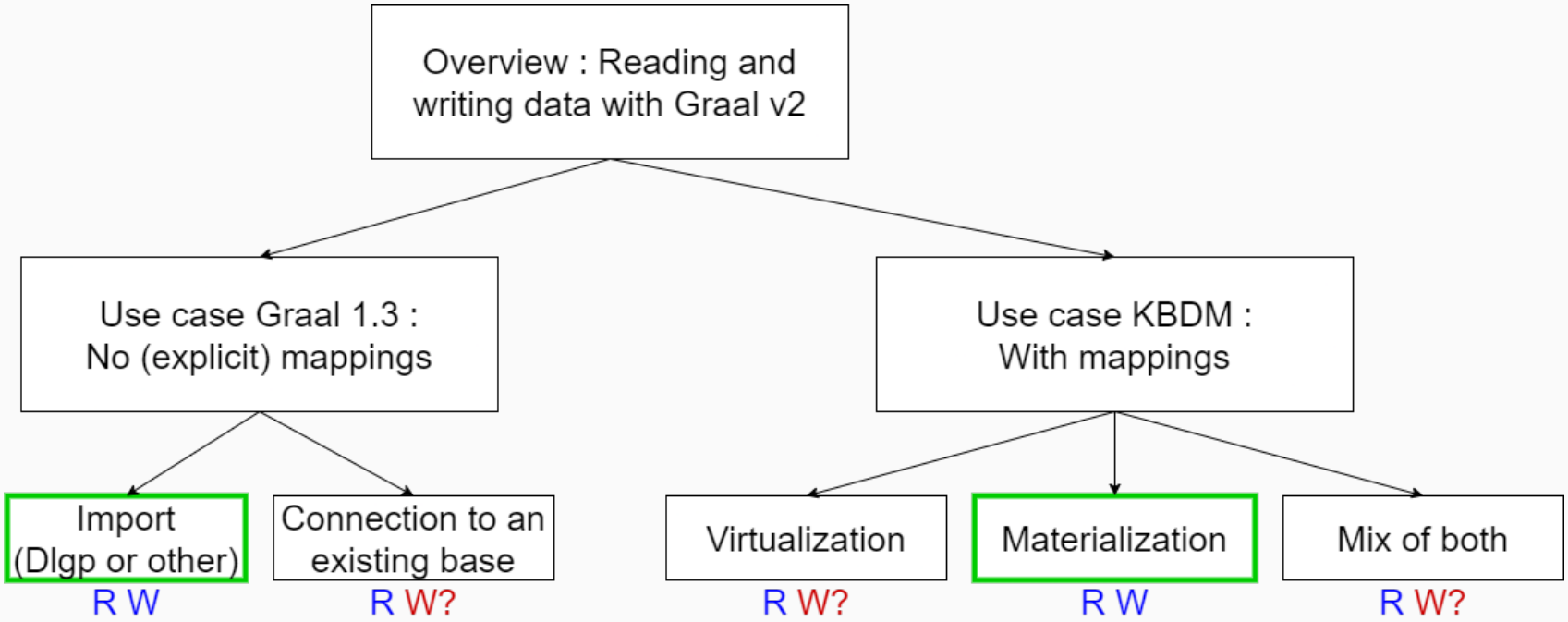
GraphIK

Introduction

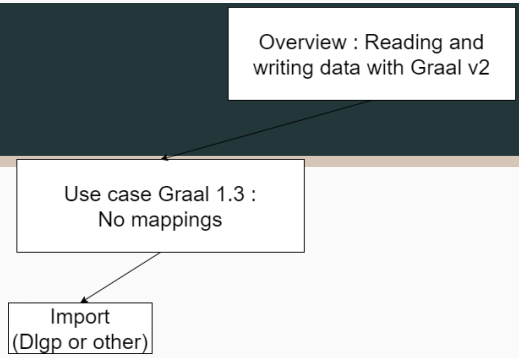
Summary

1. Use case
   - Overview
   - What can currently be done with Graal v2
2. Focus on the Chase
3. Focus on the external storage
4. Quick talk about performances
5. Development perspectives

Use case - Overview



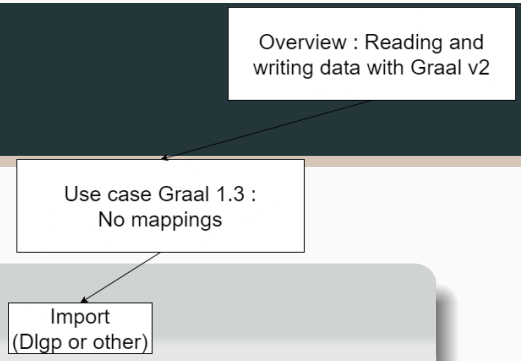
Use case - Chase



Use case 1 : Chase on Graal v2 native storage

1. Load facts and rules from Dlgp file
2. Store the facts in a native Graal data structure
3. Read (evaluate queries such as rules body)
    - Native Graal algorithm (backtrack; atom by atom)
4. Write
    - Native Graal algorithm from data structure

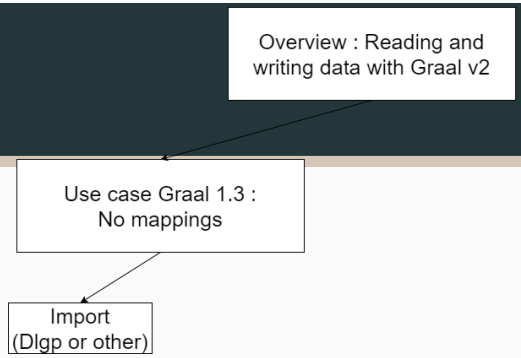
Use case - Chase



Use case 2 : Chase on a DBMS

1. Load facts and rules from Dlgp file
2. Store the facts in a database
   - Database and schema handled by Graal v2
3. Read
   1. Native Graal algorithm (backtrack; atom by atom)
   2. (Conjunctive) Queries translated to DBMS native query
4. Write
   1. Native Graal algorithm (atom by atom)
   2. Rules translated to DBMS native update

Use case - Chase



Use case 2bis : Reuse an existing Graal v2 database

1. Connect to an existing Graal v2 database
2. Load (necessary) metadata
3. See use case 2

Use case - Chase

Use case 3 : Chase with a federation

- Connect to multiple (non-Graal v2) datasource with mappings
- Materialize the mappings into a Graal v2 handled storage system
- See previous use cases

Zoom on the Chase

Goals

- Research
  - Compare different chase algorithms
  - Implement a new approach
- Applications
  - Offer the best chase version according to the scenario

Solution

Split the chase algorithm in different modules, being able to combine modules together

Zoom on the Chase

Integration work and it's difficulties

- Start from Guillaume's internship work on Graal 1.3
  - oriented towards in-memory native storage
- Adapt to the new objects and architecture
- Add needed implementations
- Ensure correct operation with external storage
- Facilitate the chase creation
- Write the documentation

Zoom on the Chase

Result - Different options

- **Rule scheduler** : Naive ; GRD
- **Rule application** : BreadthFirst ; Parallel ; DirectSQL
- **Trigger computing** : Naive ; Semi-Naive ; Two Step
- **Trigger checking** : AlwaysTrue ; Oblivious ; Semi-Oblivious ; Restricted
- **Skolem** : Fresh variable ; Body ; Frontier ; Frontier by piece
- **Halting conditions** : Facts created at previous step ; Has rules to apply ; Limit number of atoms ; Limit number of steps ; Timeout
- **Treatment** : Rule split ; Debug

Zoom on the Chase

```
FactBase fb = new SimpleInMemoryGraphStore(atoms);
RuleBase rb = new RuleBaseImpl(rules);

ChaseBuilder.defaultBuilder(fb, rb, termfactory)
.useGRDRuleScheduler()
.useParallelApplication()
.useSemiNaiveComputer()
.useRestrictedChecker()
.useGenericFOQueryEvaluator()
.debug()
.build()
.get()
.execute();
```

Zoom on the SQL storage

Goals

- Research
  - Handle larger dataset (may not fit in memory)
  - Compare different storage systems and approaches
- Applications
  - Data persistence
  - (re)Use existing storage/DBMS

Solution

Split the external storage in different modules, being able to combine modules together

Zoom on the SQL storage

Integration work and it's difficulties

- Start from Clement's work and Renaud's internship on Graal 1.3
- Adapt to the new objects and architecture
- Upgrade drivers versions
- Split the implementation in different modules
- Add needed implementations
- Ensure compatibility with Graal v2 (chase, backtrack, ...)
- Facilitate the creation and connection to external storage systems
- Write the documentation

Zoom on the SQL storage

Result - Different options

- **Driver** : SQLite ; HSQLDB ; PostGres ; MySQL
- **Strategy** : AdHocSQL

p(a, X0), p(X0, a), q(a)

predicates		
label	*arity*	*table*
p	2	pred0
q	1	pred1

terms	
label	*type*
X	v
a	c
X0	v

pred0	
TERM0	*TERM1*
a	X0
X0	a

pred1
TERM0
a

Zoom on the SQL storage

```
SQLDriver driver = new PostgreSQLDriver("localhost", "postgres", "postgres", "admin");
SQLStorageStrategy strategy = new AdHocSQLStrategy(driver);
SQLWrapper database = new SQLWrapper(driver, strategy, termfactory, predicatefactory);

database.addAll(atoms);

RuleBase rb = new RuleBaseImpl(rules);

ChaseBuilder.defaultBuilder(database, rb, termfactory)
.useGRDRuleScheduler()
.useNaiveComputer()
.useObliviousChecker()
.useSmartFOQueryEvaluator()
.useDirectSQLApplication()
.debug()
.build()
.get()
.execute();
```

Quick talk about performances

String usage

- Strings are used as names for predicates and terms
- In Graal 1.3, these strings were used in comparison and hash of the objects
- In Graal v2, a new approach is possible : use an identifier for comparison and hash

String performances in practice

Small experiments done with the chase in memory

- In Graal 1.3, string operations were most of the execution time
- In Graal v2, these operations are not using strings anymore
    - At least while we stay in memory ...

Perspectives

Octobre	Novembre	Décembre	Janvier	Février	Mars	Avril	Mai	Juin	Juillet	Août
Driver triplestore	Pont BC v1.3	Parseur DLGP				Refonte Backward Chaining			Améliorer la gestion des fédérations (médiateur, ...)	
Profiler & tests unitaires	Pont Kiabora		Fonctions & prédicats calculés				Compilation	Extention Kiabora	Interfaçage Vlog	
Finaliser l'utilisation de stockage externe	Syntaxe de mappings & parseur									
	Choix virtualisation / matérialisation							Tests de performance des chases avec les différents stockages		
BRunner - chase	BRunner - storage				BRunner - fonctions			BRunner - BC		
Documentation					Documentation			Documentation		
				Mise en avant de la plateforme - Stage(s) de master						
	Release v1				Release v2			Release v3		

Hash code

Why do we use hash

Hash is used to store the objects in *temporary* data structures in memory during algorithms.
An example is a Map from variables to terms that represent a substitution.
Entries of the map are variables identified by the corresponding hash code

How is hash code computed in java ?

The hash code for a String object is computed as :
$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$
s[i] is the i^{th} character of the string
n is the length of the string
The default hash code for an object is derived from the memory address

Use case B - Implicit mappings

- Connection to an existing source
  - We do not handle the schema
- No problem (in theory) for reading
- What happen when we want to write ? **typing problems**
  - RDF triplestore : everything can be translated
  - SQL
    - How to write an existential into a date field ?
    - How to write a string into a number field ?
    - Could be handled by keeping some inference in (another) local base
  - JSON
    - Key representing the predicate
    - Maybe no typing problems because there is no schema
    - How do we handle different types in arrays (ie : [”ciao”,42])

Use case C - Mappings virtualization

1. Every source stay out of Graal and we only read. Queries are on the source vocabulary
    - Possible with Graal backtrack
    - These queries can be obtained by rewriting (once we have BC in Graal v2)
2. We need to write
    - Need mappings and rights for writing

Use case E - Mappings materialization

- Create a local base (that don't require mappings)
  - We copy only part of the source (what is covered by mappings)
- Queries are on both vocabularies (ontology and source)
  - A predicate P on the ontology is associated to local :p (invisible to the user)
  - We therefore only have queries on the vocabulary of the sources