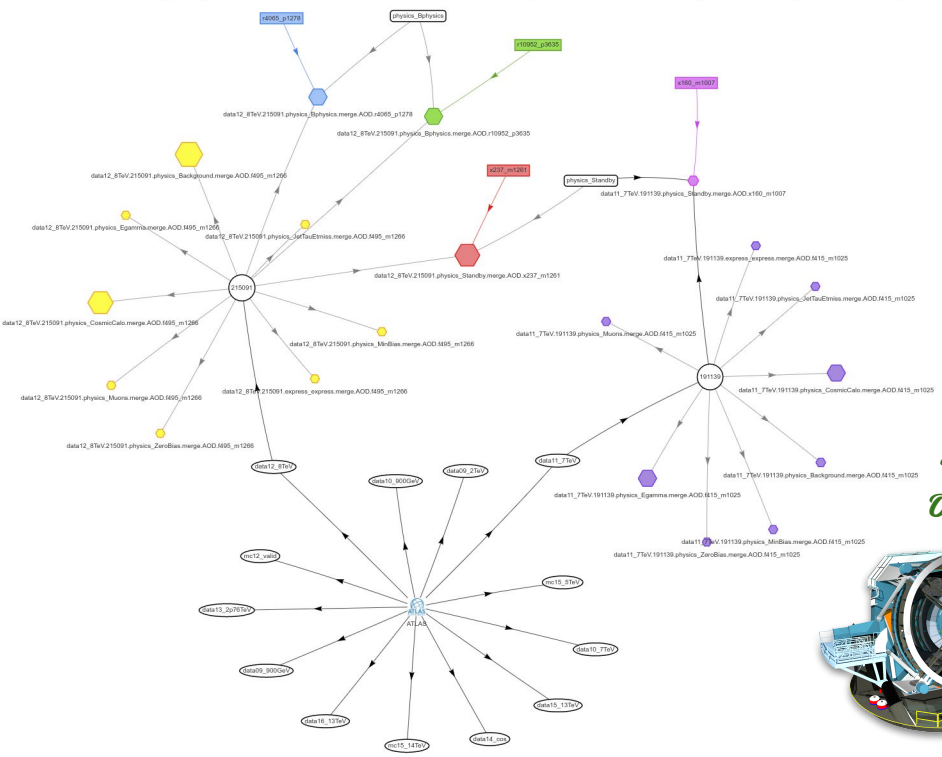


Hybrid Databases & Multilanguage Frameworks

Two ways
to prevent technology lock-up and
to profit from multiple technologies in their fields of strength.

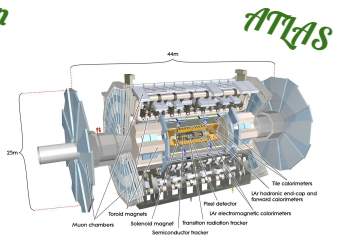


➤ Databases

- Current status of the HEP storage
- Graph databases
- Hybrid solutions
- Real-life examples
- Graph & Hybrid Databases for HEP

➤ Languages

- Ideal Multilanguage Application
- JVM Multilanguage Environment
- GraalVM
- Plurality World
- Future of programming



Julius Hrivnac, IJCLab
Journées R&T
Strasbourg, 6-8/Nov/2023



Graph & Hybrid Databases



Our Data - status

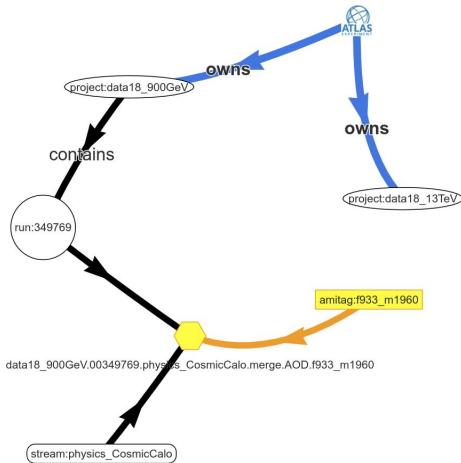
- Traditional data structures in HEP & Astronomy:
 - tuples, tables, datagrams
 - trees
 - nested tuples (trees of tuples)
 - relational (SQL-like)
 - files
- Schema-based or schema-less
- **But many of HEP data are graph-like & schema-less**
 - **Entities with relations**
- Not handled by standard tree-ntuple storage
 - Relations should be added and interpreted outside storage
- Not well covered by relational (SQL) databases
 - We need to add new relations, not covered by schema
- Difficult to manage by Object Oriented (OO) databases or serialisation
 - Problem to distinguish essential relations from volatile ones



Graph Databases

- Storing Graphs in a database
- **Graph = (Vertexes, Edges), $G = (V, E)$**
- **Vertices and Edges have properties**

- Graph databases have existed for a long time
 - Matured only recently thanks to Big Data & AI (Graph NN)
 - Very good implementations & (de-facto) standards available
 - Rapid evolution
- **Moving essential structure from code to data**
 - Together with migration from imperative to declarative semantics
 - Things don't **happen**, but **exist**
 - Structured data with relations facilitates **Declarative Analyses**
- **Data elements appear in a Context**
 - Which simplifies understanding, analyses and processing
- The difference between SQL and Graph database is similar as between Fortran and C++/Java
 - On one side, a rigid system, which can be very optimized
 - On the other side, a flexible dynamical system, which allows expressing of complex structures
- Graph database is a synthesis of OO and SQL databases
 - Expressing web of objects without fragility of OO world
 - Capturing only essential relations, not an object dump





Graph Databases - APIs

➤ Direct manipulation of Vertices and Edges

- Always available from all languages
- Doesn't use full graph expression power

➤ Cypher (or GQL)

- Pure **declarative**
- Inspired by SQL and OQL
 - But applied to schema-less database
- Available to all languages via JDBC-like API
 - Semantic mismatch, passed as String
 - There is a wall between coder and database, with a thin tunnel, only Strings can pass
- Coming from Neo4J
 - Accepted as a standard
 - Neo4J can be also used with Gremlin

```
MATCH (a:run)-[:has]->(b:dataset)
WHERE a.rnumber = 98765
RETURN b.name
```

*All three methods can be used to access the same database
(e.g. using Cypher for query and Gremlin for traversal).*

➤ Gremlin

- **Functional** syntax
- Originated from *Groovy*, but available to many languages supporting functional programming Integrated in the language
- Supported by the generic Tinkerbox framework
- Well integrated in the host language

```
g.V().has('run', 'rnumber', 98765)
.out('has')
.values('name')
```

Standards & Choices



- De-facto standard language/api: **Gremlin**
 - Gremlin is a functional, data-flow language to **traverse a property graph**. Every Gremlin traversal is composed of a sequence of (potentially nested) steps. A step performs an atomic operation on the data stream. Every step is either a *map*-step (transforming the objects in the stream), a *filter*-step (removing objects from the stream), or a *sideEffect*-step (computing statistics about the stream).
 - Gremlin supports **transactional & non-transactional** processing in **declarative** or **imperative** manner.
 - Gremlin can be expressed in all languages supporting function composition & nesting.
- Commonly used framework: **TinkerPop**
- Leading implementation: **JanusGraph**
 - Supported storage backends: Cassandra, **HBase**, Google Cloud, Oracle BerkeleyDB
 - Supported graph data analytics: Spark, Giraph, Hadoop
 - Supported searches: Elastic Search, Solr, Lucene
 - Growing popularity of Neo4J
- Possible visualisation: **visj.org**
 - Many others exist



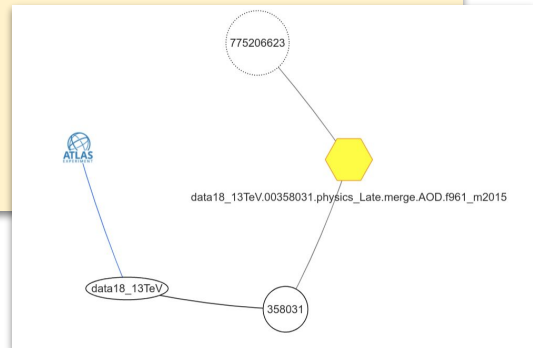
Gremlin Syntax

- Functional syntax
- **Functional & navigational semantics**
- Very intuitive, **no special syntax needed** (using existing functional syntax), easy integration.
- Database just accessed as objects with structure and relations.
 - Nested collections with links.
- Can use functional API (streams) and Lambda.
- **No semantic mismatch.**
 - Using one language.
- Came from **Groovy**
 - Groovy = Java + syntactic sugar, useful for scripting + suitable for functional code.
- Gremlin clients exist for many languages
 - **Java, Groovy, Python C#, JS, C++**, Clojure, Elixir, Go, Kotlin, Haskell, PHP, Ruby, Rust, Scala,...
- Both **search** and **traversal** steps.
- Search steps can be boosted by indexes.
- Functions can be loaded on server for faster execution.

Functional syntax with additional navigational semantics !

```
# add a vertex 'experiment' with the name 'ATLAS'
g.addV('experiment').property('ename', 'ATLAS')
# add edges 'owns' from all vertices 'project' to vertex 'experiment' 'ATLAS'
g.V().hasLabel('project')
    .addE('owns')
    .from(g.V()
        .hasLabel('experiment')
        .has('ename', 'ATLAS'))

# show datasets with more events or number of events in an interval
g.V().has("run", "number", 358031)
    .out() # dataset
    .has("nevents", gt(7180136))
    .values("name", "nevents")
g.V().has("run", "number", 358031)
    .out() # dataset
    .has("nevents", inside(7180136, 90026772))
    .values("name", "nevents")
```





Performance

- Requests in general in three phases
 - First **search** of the initial entry point (event, dataset, run,...)
 - Could be optimised
 - Natural order
 - Indexes
 - Elastic Search
 - Spark
 - More hierarchical navigation
 - Then **navigation** on the graph
 - Very fast
 - And finally **accumulation** of results
- Data can still be accessed directly, without Graph Database API
 - So with the same performance as non-GraphDB
 - Navigational step (instead of sub-search) can only speed it up
- In general:
 - Very fast retrieval
 - Slower import
 - Because import creates structures
 - Which are used in retrieval (simpler & faster)
 - Very slow deletion

Performance Example



75% of the time is spend by the entry point search, following graph traversal is very fast.

This was the first request, the second one will be cca 10x faster (even on different event).

```
gremlin> el(358031, 775206623, g).profile()
=>Traversal Metrics
Step                                                                 Count  Traversers    Time (ms)  % Dur
-----
JanusGraphStep([], [~label.eq(event), enumber.eq...           1      1          204.805    75.74
  \_condition=(~label = event AND enumber = 775206623)
  \_isFitted=true
  \_query=multiKSQ[1]@2147483647
  \_index=event:enumber:u
  \_orders=[]
  \_isOrdered=true
  optimization
  optimization
  backend-query
  \_query=event:enumber:u:multiKSQ[1]@2147483647
JanusGraphVertexStep(IN, [keeps], vertex)                       1      1          25.560     9.45
  \_condition=type[keeps]
  \_isFitted=true
  \_vertices=1
  \_query=org.janusgraph.diskstorage.keycolumnvalue.SliceQuery@b3a55b7f
  \_orders=[]
  \_isOrdered=true
  optimization
  backend-query
  \_query=org.janusgraph.diskstorage.keycolumnvalue.SliceQuery@b3a55b7f
JanusGraphVertexStep(IN, [fills], vertex)                       1      1          10.388     3.84
  \_condition=type[fills]
  \_isFitted=true
  \_vertices=1
  \_query=org.janusgraph.diskstorage.keycolumnvalue.SliceQuery@b3a605c1
  \_orders=[]
  \_isOrdered=true
  optimization
  backend-query
  \_query=org.janusgraph.diskstorage.keycolumnvalue.SliceQuery@b3a605c1
HasStep([rnumber.eq(358031)])                                   1      1           13.129     4.86
SelectOneStep(last,e)                                          1      1           0.993     0.37
NoOpBarrierStep(2500)                                          1      1           0.159     0.06
JanusGraphPropertiesStep([guid], value)                        2      2           14.800     5.47
  \_condition=type[guid]
  \_isFitted=true
  \_vertices=1
  \_query=org.janusgraph.diskstorage.keycolumnvalue.SliceQuery@b11f98a7
  \_orders=[]
  \_isOrdered=true
  optimization
  NoOpBarrierStep(2500)                                         2      2           7.478     2.81
  NoOpBarrierStep(2500)                                         2      2           0.568     0.21
  >TOTAL                                                         -      -          270.406     -
```



Graph databases 'problems'

- Rather slow insert/import
- Very slow cleaning (memory management)
- Anarchical Edges creation can easily create un-managable structures
- Persistent and volatile data are not well separated
- Not very optimised for mass, parallel processing of huge homogeneous tables
 - Unknown schema, chaotic relations,...
- Advanced Gremlin is very powerful, but can be rather cryptic
 - Multi-dimensional functional syntax
 - Hard to understand what's going on
 - Which parts are executed and which are navigational (on the storage)
 - Which parts are evaluated lazily

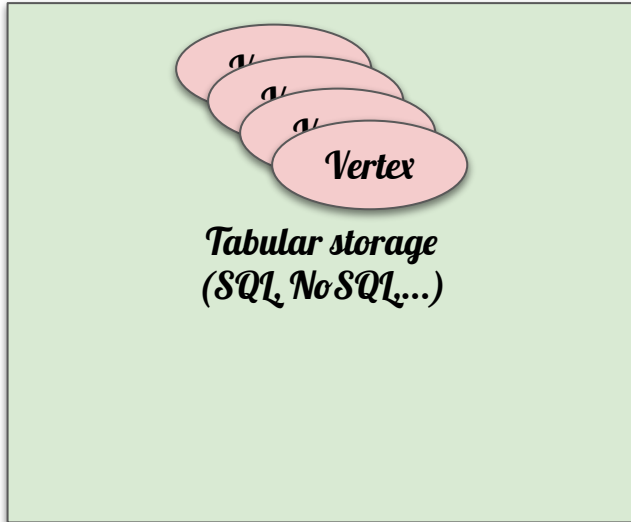


Hybrid Solutions

- Store unstructured (raw) data in table-like storage (SQL, NoSQL)
 - Suitable for intensive, parallel processing (Spark,...)
 - Interpretable as **Datagrams**-like apis
- Express persistent **structure** as a **Graph**
- Allow for ad-hoc (a'priority volatile) Graph relations
 - Possibly in separated (but connected) graphs
 - Playgrounds, Whiteboards,...
- Connect everything behind the **common API**



Hybrid Solutions - Graph View



- **Interpret existing tabular data as Vertices in a Graph**
- Add additional Edges to express structures
- Requires full-featured rather generic implementation of the Graph storage
 - Difficult to implement
 - Doesn't exist
 - Most Graph implementation use tabular store as a backend, but impose their own schema



Hybrid Solutions - Graph Envelope



- **Make an enhanced version of Vertex with additional methods to fill it from external tabular storage**
- Feasible, has been implemented
- Logical problems:
 - Consistency between (already copied) Vertex and original data
 - Search semantics
- Unpredictable performance
 - Don't know where actually are data and whether will be copied or accessed remotely

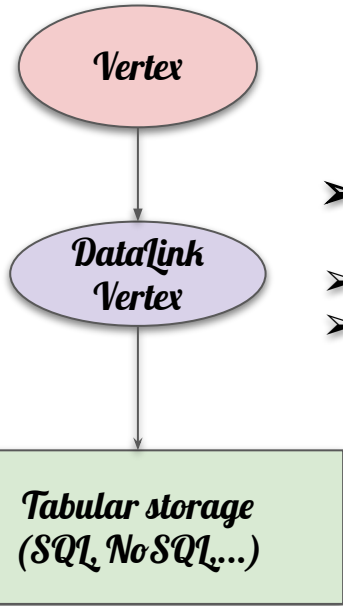
*Tabular storage
(SQL, NoSQL,...)*

```
// Create an alert vertex
v = g.addV().property('lbl', 'alert')
// Dress it as (a subtype of) Hertzex (= HBase backed Vertex)
// which _is_a_ Vertex so it has all Vertex properties
h = Hertzex.enhance(v)

// Create a new alert vertex (connect to HBase data later)
a = Alert.getOrCreate('ZTF19acmbwur_2458789.0311458', g, false);
// Create a new alert vertex (and connect to HBase data)
a = Alert.getOrCreate('ZTF19acmbwur_2458789.0311458', g, true);
```



Hybrid Solutions - Bridges



- **Make special kind of DataLink Vertex representing relations to external data (in any storage)**
- Those Vertexes can be attached to any Vertex
- Advantage:
 - Easy to implement
 - Transparent logic
 - Works between any pair of databases with any technology
 - We can even connect to Graphs like that

```
// Create DataLink Vertexes with associated data in another database (Phoenix/SQL, Graph, HBase,...)
w1 = g.addV().property('lbl', 'datalink').property('technology', 'Phoenix').property('url', 'jdbc:phoenix:itthdp2101.cern.ch:2181' ).property('query', "...")
w2 = g.addV().property('lbl', 'datalink').property('technology', 'Graph' ).property('url', 'hbase:188.184.87.217:8182:janusgraph').property('query', "...")
w3 = g.addV().property('lbl', 'datalink').property('technology', 'HBase' ).property('url', '134.158.74.54:2183:ztf:schema' ).property('query', "...")
// Connect DataLink to any Vertex
theVertex.addEdge('externalData').to(w)
// Get associated data
externalData = Lomikel.getDataLink(w)
```

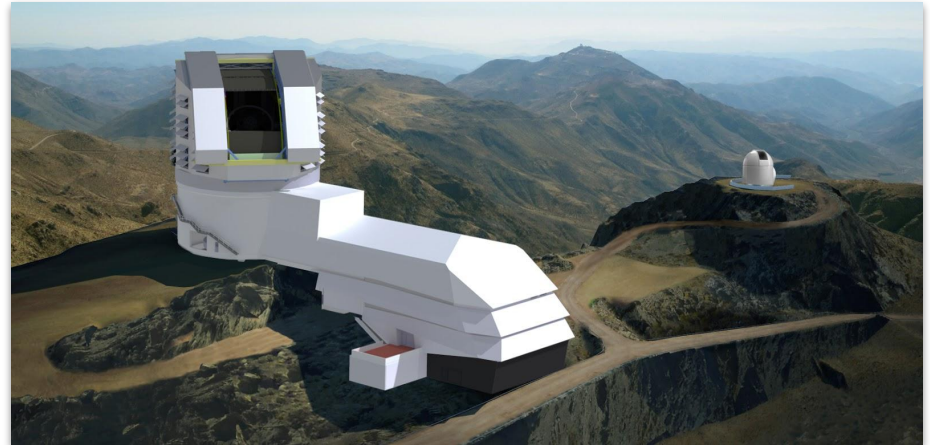
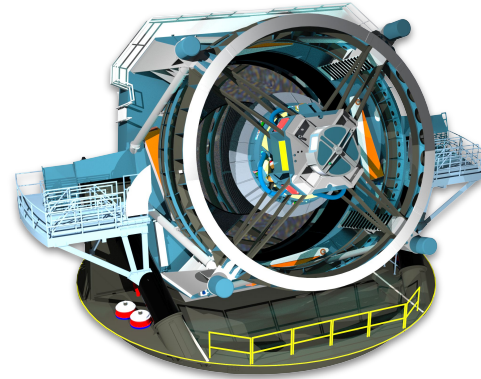


Real-life examples

Funk/LSST



- **Observatoire Vera C. Rubin for Legacy Survey of Space and Time (LSST)**
- Looking for **changing/moving objects**
- Camera 8.4 m, 3.2 Gpixel in Chili
- 10 millions alertes, 20 TB nightly
- 500 PB in 10 years (3 PB of alerts)
- **Alertes send over the world via network of 'brokers'**
- Commissioning in 2023, production in 2024





FINK, a new generation of broker for the LSST community

Anais Möller^{1*}, Julien Peloton^{2†}, Emille E. O. Ishida^{1‡},
 Chris Arnault², Etienne Bachelet³, Tristan Blainneau⁴, Dominique Boutigny⁴,
 Abhishek Chauhan⁵, Emmanuel Gangler¹, Fabio Hernandez⁶, Julius Hrivnac²,
 Marco Leoni^{2,7}, Nicolas Leroy², Marc Moniez², Sacha Pateyron², Adrien Ramparison²,
 Damien Turpin⁸, Réza Ansari², Tarek Allam Jr.^{9,10}, Armelle Bajat¹², Biswajit Biswas^{1,13},
 Alexandre Boucaud¹⁴, Johan Bregeon¹⁵, Jean-Eric Campagne², Johann Cohen-Tamugi^{16,1},
 Alexis Coleiro¹⁴, Damien Dornic¹⁷, Dominique Fouchez¹⁷, Olivier Godet¹⁸, Philippe Gris¹,
 Sergey Karpov¹², Ada Nebot Gomez-Moran¹⁹, Jérémy Neven²,
 Stephane Plaszczyński², Volodymyr Savchenko²⁰, Natalie Webb¹⁸

¹ Université Clermont Auvergne, CNRS/INSP, LFC, F-63000 Clermont-Ferrand, France
² Université Paris-Saclay, CNRS/INSP, ICLab, Orsay, France
³ Las Cumbres Observatory, 6710 Cortona Drive, suite 102, Gillett, CA 93117, USA
⁴ Université Grenoble-Alpes, Université Savoie Mont Blanc, CNRS/INSP Laboratoire d'Annecy-le-Vieux de Physique des Particules, France
⁵ Department of Ocean Engineering and Naval Architecture, Indian Institute of Technology Kharagpur, West Bengal, 721302, India
⁶ CNRS, CC-INSP3, 21 avenue Pierre de Coubertin, CS70002, 69627 Villeurbanne cedex, France
⁷ Direction des Systèmes d'Information, Université Paris Sud, 91405 Orsay Cedex, France
⁸ Université Paris-Saclay, CNRS, CEA, Département d'Astrophysique, Instrumentation et Modélisation de Paris-Saclay, Gif-sur-Yvette, France
⁹ Centre for Data Intensive Science, Department of Physics and Astronomy, University College London, London WC1E 6BT
¹⁰ Mullard Space Science Laboratory (MSSL), Department of Space and Climate Physics, University College London, Surrey RH5 6NT, UK
¹¹ CEICO, Institute of Physics, Czech Academy of Sciences, Prague, Czech Republic
¹² Department of Physics, Birla Institute of Technology and Science, Pilani, Pilani Campus, Rajasthan, 333031, India
¹³ Université de Paris, CNRS, AstroParticule et Cosmologie, F-75013, Paris, France
¹⁴ CNRS-INSP, Laboratoire de Physique Subatomique et de Cosmologie (LPSC), Grenoble
¹⁵ LUPM, Université de Montpellier et CNRS, 34095 Montpellier Cedex 05, France
¹⁶ Aix Marseille Université, CNRS/INSP, CPPM, Marseille, France
¹⁷ IRAP UPS/CNRS/CNRS, 9 avenue du colonel Roche 3028 Toulouse Cedex 04, France
¹⁸ Observatoire Astronomique de Strasbourg, Université de Strasbourg, CNRS UMR 7550, 11 rue de l'Observatoire, 67000 Strasbourg, France.
¹⁹ SDSC, Department of Astronomy, University of Geneva, Chemin d'Ecogia, 16 CH-1200 Versoix, Switzerland

23 September 2020

ABSTRACT

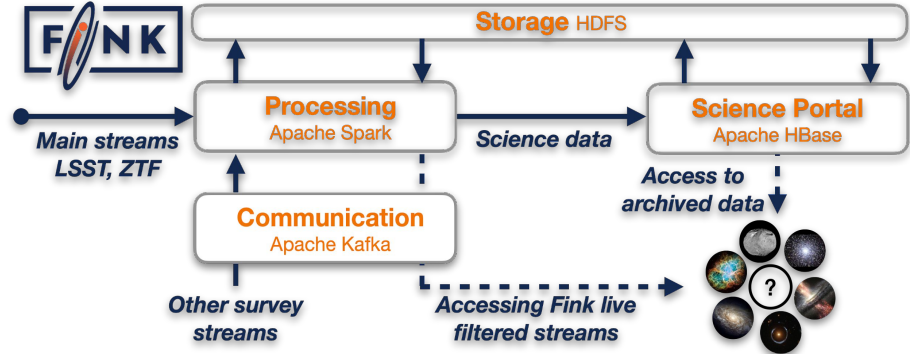
FINK is a broker designed to enable science with large time-domain alert streams such as the one from the upcoming Legacy Survey of Space and Time (LSST). It exhibits traditional astronomy broker features such as automated ingestion, enrichment, selection and redistribution of promising alerts for transient science. It is also designed to go beyond traditional broker features by providing real-time transient classification which is continuously improved by using state-of-the-art Deep Learning and Adaptive Learning techniques. These evolving added values will enable more accurate scientific output from LSST photometric data for diverse science cases while also leading to a higher incidence of new discoveries which shall accompany the evolution of the survey. In this paper we introduce FINK, its science motivation, architecture and current status including first science verification cases using the Zwicky Transient Facility alert stream.

Key words: surveys – methods: data analysis – software: data analysis – transients: gamma-ray bursts – gravitational lensing: micro – transients: supernovae

* E-mail: anais.moller@clermont.in2p3.fr

† E-mail: peloton@lal.in2p3.fr

- Using Apache Spark & Big Data / NoSQL
 - Hadoop & JanusGraph for storage
- Fink is one of the official LSST brokers





Fink Science Portal (J. Peloton)



Fink Science portal 0.4 Explorer Xmatch API Info



Explore Fink historical data

Browse all alert data collected and processed by Fink

Explorer

Cross-match with Fink data

Upload your catalog and cross-match against Fink alert data

Xmatch

Explore the Graph database

Visualise links between alerts and detect patterns in Fink data

Grafink (Not yet available)



Fink Science portal 0.4 Explorer Xmatch API Info



objectID - Enter a valid object ID or choose another query type

- ZTF Object ID
- Conesearch
- Date Search
- Class search
- SSO search

Fink Science portal 0.4 Explorer Xmatch API Info

SSO - 9247

Info Table Sky map

Add more fields to the table

iobjectid	lra	ldec	vlastdate	vclassification	indethist
ZTF21aasslmh	332.4415668	10.2270298	2021-04-05 12:32:48.002	Solar System	1
ZTF21aasslib	332.4401292	10.226322	2021-04-05 12:25:48.996	Solar System	1
ZTF21aasslcq	332.4393507	10.2260407	2021-04-05 12:22:20.004	Solar System	1
ZTF21aasslct	332.137864	10.0876954	2021-04-04 12:33:54.996	Solar System	1
ZTF21aasslhux	332.1371574	10.0873636	2021-04-04 12:30:25.001	Solar System	1
ZTF21aasslhp	332.13572	10.0866677	2021-04-04 12:23:26.998	Solar System	1
ZTF21aasslhp	332.13572	10.0866677	2021-04-04 12:23:26.998	Solar System	1



Fink Science Portal (J. Peloton)



Fink Science portal 0.4 Explorer Xmatch API Info

SSO 9247

Info Table Sky map

Aladin

Fov: 180"

Hit the Aladin Lite fullscreen button if the image is not displayed (we are working on it...)

Fink Science portal 0.4 Explorer Xmatch API Info

ZTF21aasslmh

Summary Supernovae Variable stars Microlensing Solar System GRB

Download ZTF21aasslmh data

Download 9247 data

Name: 9247
Orbit type: Unclassified (mostly Main Belters)

Properties from MPC
number: 9247
period (year): 5.5
a (AU): 3.1155337
q (AU): 2.9028206
e: 0.068275
inc (deg): 26.4613
Omega (deg): 213.0870065
argPeri (deg): 251.87212
tPeri (MJD): 60065.39929000005
meanAnomaly (deg): 284.89647
epoch (MJD): 59200.0
g: 0.15
neo: 0

MPC JPL

Fink Science portal 0.4 Explorer Xmatch API Info

ZTF21aassamj

Different views Summary Supernovae Variable stars Microlensing Solar System GRB

ObjectID: ZTF21aassamj
Fink class: Early SN candidate

Supernova classifiers
SN Ia score: 0.02
SNe score: 0.95
Early SN classifier
RF score: 0.97

Variability (DC magnitude)
Rate g-r (last): -0.03 mag/day
Rate g (last): -0.08 mag/day
Rate r (last): -0.10 mag/day

Extra properties
Classstar: 0.98
Detection in the survey: 10
DL Real bogus: 1.00

TNS OAC
SIMBAD NED SDSS

Download ZTF21aassamj data

Extra properties

Fink Science portal 0.4 Explorer Xmatch API Info

ZTF19acnjwgm

Summary Supernovae Variable stars Microlensing Solar System GRB

Science Template Difference

ObjectID: ZTF19acnjwgm
Fink class: SN candidate

General properties
Date: 2020-01-29 03:18:40.003
RA: 36.6004139 deg
Dec: 28.5079626 deg

Variability (DC magnitude)
Rate g (last): 0.04 mag/day
Rate r (last): 0.03 mag/day

Neighbourhood
SIMBAD: Transient
MPC: null
Distance (PS1): 5.40 arcsec
Distance (Gaia): 13.16 arcsec
Distance (ZTF): 1.78 arcsec

TNS OAC
SIMBAD NED SDSS

Download ZTF19acnjwgm data

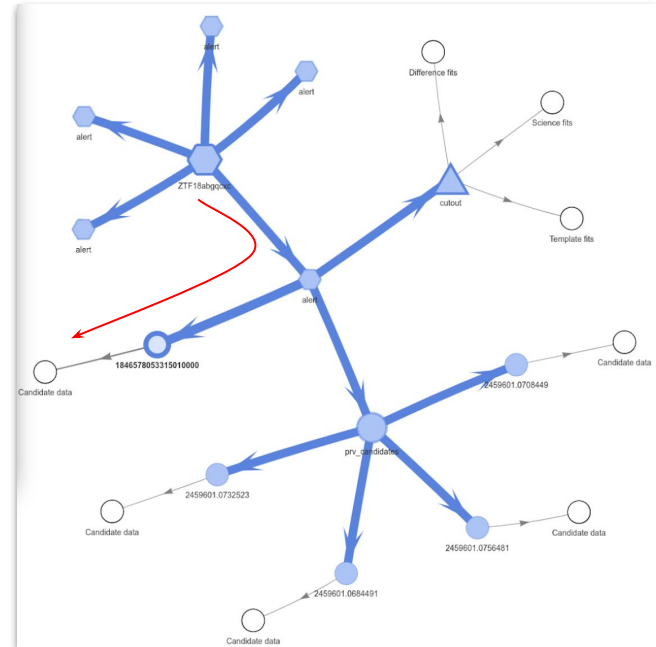
Difference magnitude views show:
Circles (●) with error bars show valid alerts that pass the Fink quality cuts. In addition, the Difference magnitude views show:

- All coming alert data are stored in HBase tables
- The alert data structure is created in the JanusGraph
 - Contains also the most important attributes
 - Has datalinks to HBase data

```

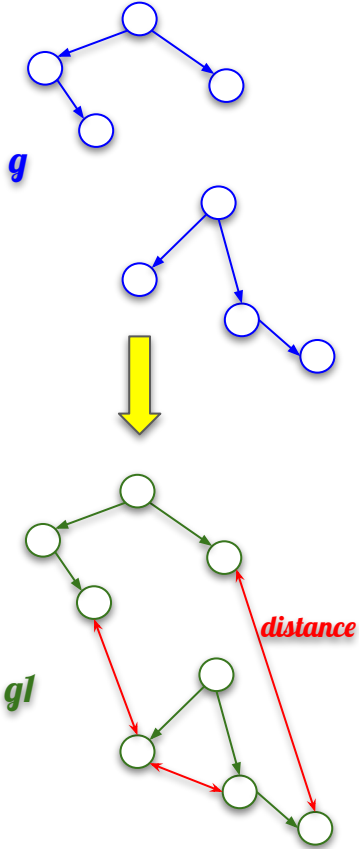
// Get data (from HBase) attached in 'candidate's
g.V().has('lbl', 'source' ).
  has('objectId', 'ZTF18abimys').out().
  has('lbl', 'alert' ).out().
  has('lbl', 'candidate' ).out().
  has('lbl', 'datalink' ).
  each {
    println(FinkBrowser.getDataLink(it))
  }

```



Funk/LSST

*Search for 'interesting' relations and store them in Graph as Edges for later analyses.
Do it in your private subgraph.*

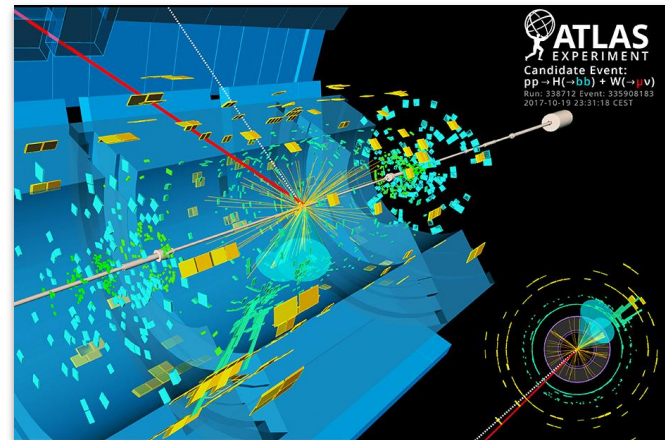
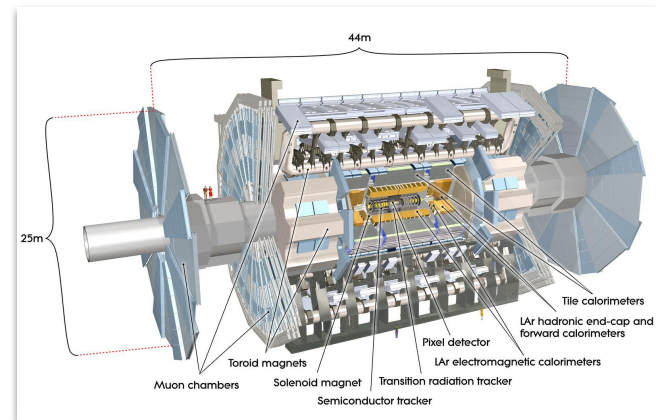


```
// Create a new personal Graph.
graph1 = Lomikel.myGraph()
// Get the entry point to the Graph traversal.
g1 = graph1.traversal()
// GremlinRecipies is a class with various useful Gremlin methods.
gr = new GremlinRecipies(g)
// Get 'source' Vertexes from the main Graph (automatically available as 'g') and
// clone them in the private Graph 'g1'.
g.V().has('lbl', 'source').each {source ->
    gr.gimme(source, g1, -1, -1)
}
// Get GremlinRecipies for the private graph 'g1'.
gr1 = new GremlinRecipies(g1)
// Find all pairs of 'candidate' Vertexes, where difference between their 'rb'
// fields is bigger or equal to 0.01.
// Connect them with the Edge 'distance' having a 'difference' property equal to
// the difference between 'rt' fields.
gr1.structured(g1.V().has('lbl', 'candidate'), 'rb[0]-rb[1]', 'rb', 0.01, 'distance', 'difference', ...)
// Get some statistics about newly created Edges.
g1.E().hasLabel('distance').values('difference').union(min(), max(), sum(), mean(), count())
```

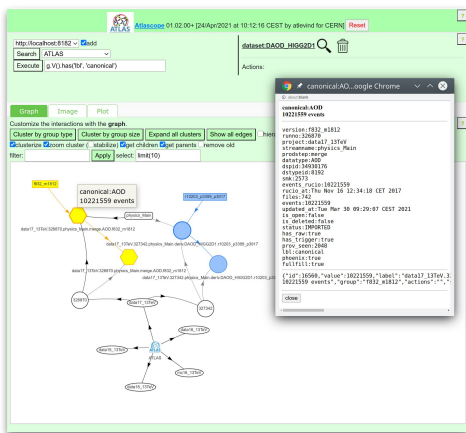
Event Index/ATLAS



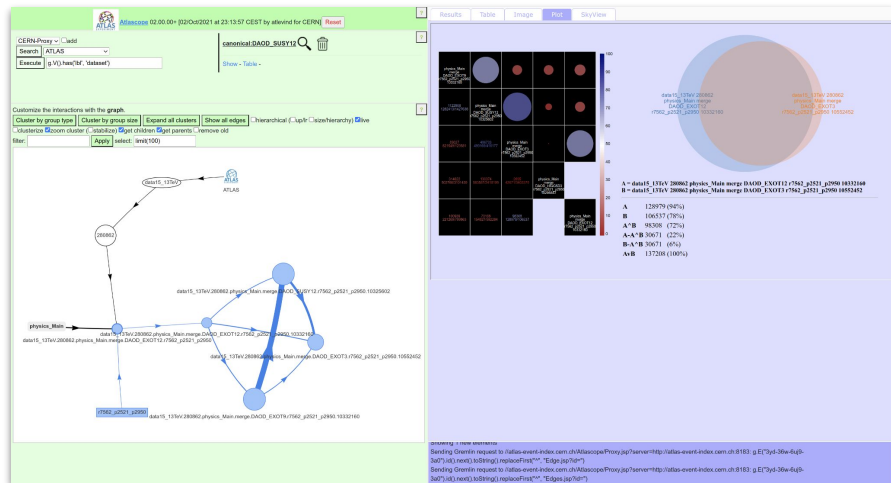
- One of the four experiments at LHC in CERN
- 38 pays
- 25 milliards of events stored with speed of 1 GB/s, ATLAS Grid contains 200 PB
- 400 sw developers
- 6 millions lines of code
- In C++, Python, Java, F90, ...



Event Index/ATLAS

ATLAS Event Index interface showing search results for 'canonical_AOD'. The interface includes a search bar, a graph visualization, and a detailed view of the selected dataset.



ATLAS Event Index interface showing search results for 'canonical_AOD'. The interface includes a search bar, a graph visualization, and a detailed view of the selected dataset.

- **Index of all ATLAS data (real and simulated)**
 - Contains more than $360 \cdot 10^9$ entries
- **Front-end for the analyses of data**
- **Migration from Run 2 implementation to new Run 3 implementation**
 - Using **Hadoop HBase** and **Phoenix** (SQL layer)
- **About 2000 accesses per day**
- **JanusGraph** for
 - Relations between objects (overlaps between datasets,...)
 - Virtual collections

Event Index/ATLAS

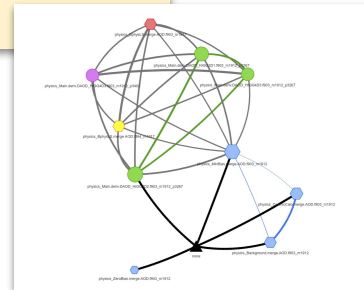
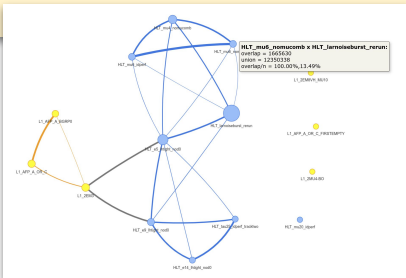
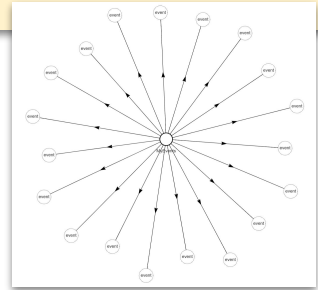
```
// Get a dataset
dataset = g.V().has('lbl', 'dataset').
  has('project', 'mc16_13TeV').
  has('streamname', 'Py8EG_SSM_Wprime500_Zprime250').
  has('prodstep', 'merge').
  has('datatype', 'EVENT').
  has('runno', 801122).
  has('version', 'e8307_e7400').next()
```

```
// Import a dataset content (from Phoenix/HBase)
Event.getOrCreate(dataset, null, 3154, g, true).each{event -> println(event.connect())}
```

```
// Find the overlap between two datasets
// (i.e. overlap Edge between dataset Vertexes)
g.V().has('lbl', 'dataset').
  has('name', 'data18_13TeV.00358031.physics_Main.deriv.DAOD_HIGG2D1.f961_m2015_p3597').
  out('overlap').
  has('dataset', 'name', 'data18_13TeV.00358031.physics_Main.deriv.DAOD_HIGG1D1.f961_m2015_p3583')
```

```
// Create new collection of events
eventsCollection = g.addV('ecollection')
  .property('name', 'MyEvents')
```

```
// Find all events satisfying certain conditions
// and connect them to the event collection
g.V().has('lbl', 'event')
  .has(...some selection...)
  .collect {
    eventsCollection.addEdge('contains', it)
  };
```





Graph Databases for Functional Programming

- Relations (edges) can be considered as functions
 - Navigation as a function execution
 - **From the user point of view, there is no difference in creating new object or navigating to it**
 - Both operation can be 'lazy'
- Functional processing and graph navigation ("*Graph Oriented Programming*") can work very well together
 - Using the same functional syntax
 - Both are realisation of Categories
 - Vertex == object, Edge == morphism
 - Functional program can be modeled as a Graphs
 - Graph data can be navigated using functions
 - Data **ready for parallel access**
- Very well implemented by Gremlin

Extending parallel-ready functional model from code to data !



Graph Databases for Deep Learning

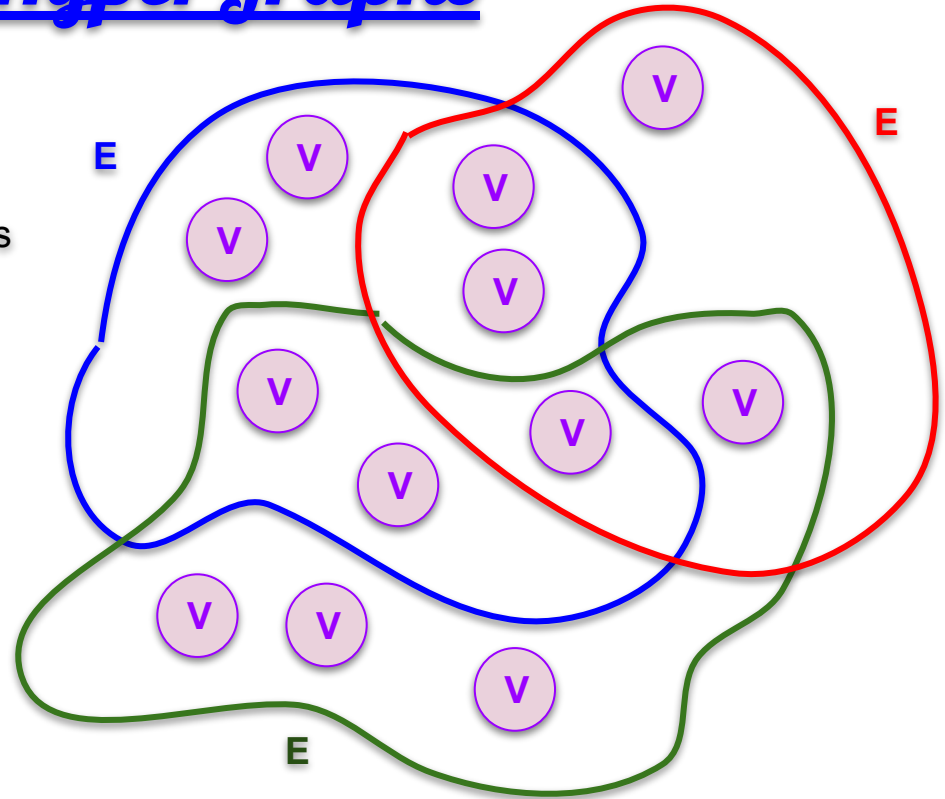
- Neural Network itself is a Graph
 - Using Graph Database to describe NN itself
- In many cases, Neural Network handles Graph data (objects with relations)
 - They can operate either on individual nodes (Node-focused tasks)
 - Or on the whole graph (Graph-focused tasks)
- GraphNN can be seen as a generalisation of ConvolutionalINN
 - Non-geometric
- Possibility to impose **constraints/knowledge** to NN
 - Inductive Bias
 - Semantic Induction

Graph Neural Networks create a Natural environment for Deep Learning !

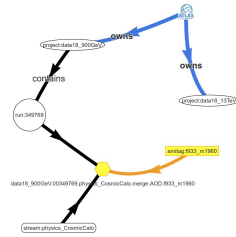


Future HyperGraphs

- Coming from AI
- Generalisation of Graph paradigme
- One Edge can connect multiple Vertices
- Can create rich structures
 - Covers all structures we already
- Can serve sophisticated algorithms
- No Open Source toolkit yet



Graph & Hybrid Databases for HEP



➤ A lot of ongoing HEP effort to make execution more structured and parallel

- Parallel programming
- Functional programming

➤ Less effort (so far) to structure the data

- More structured data => simpler and faster access

➤ Graph Database advantages

- More transparent code
 - Stable data structure is handled in the storage layer
- Suitable for **Functional Style** and **Parallelism**
- Suitable for **Deep Learning**
- Suitable for **Declarative Analyses**
- Can help with **Analysis Preservation**
- Language & Framework neutral

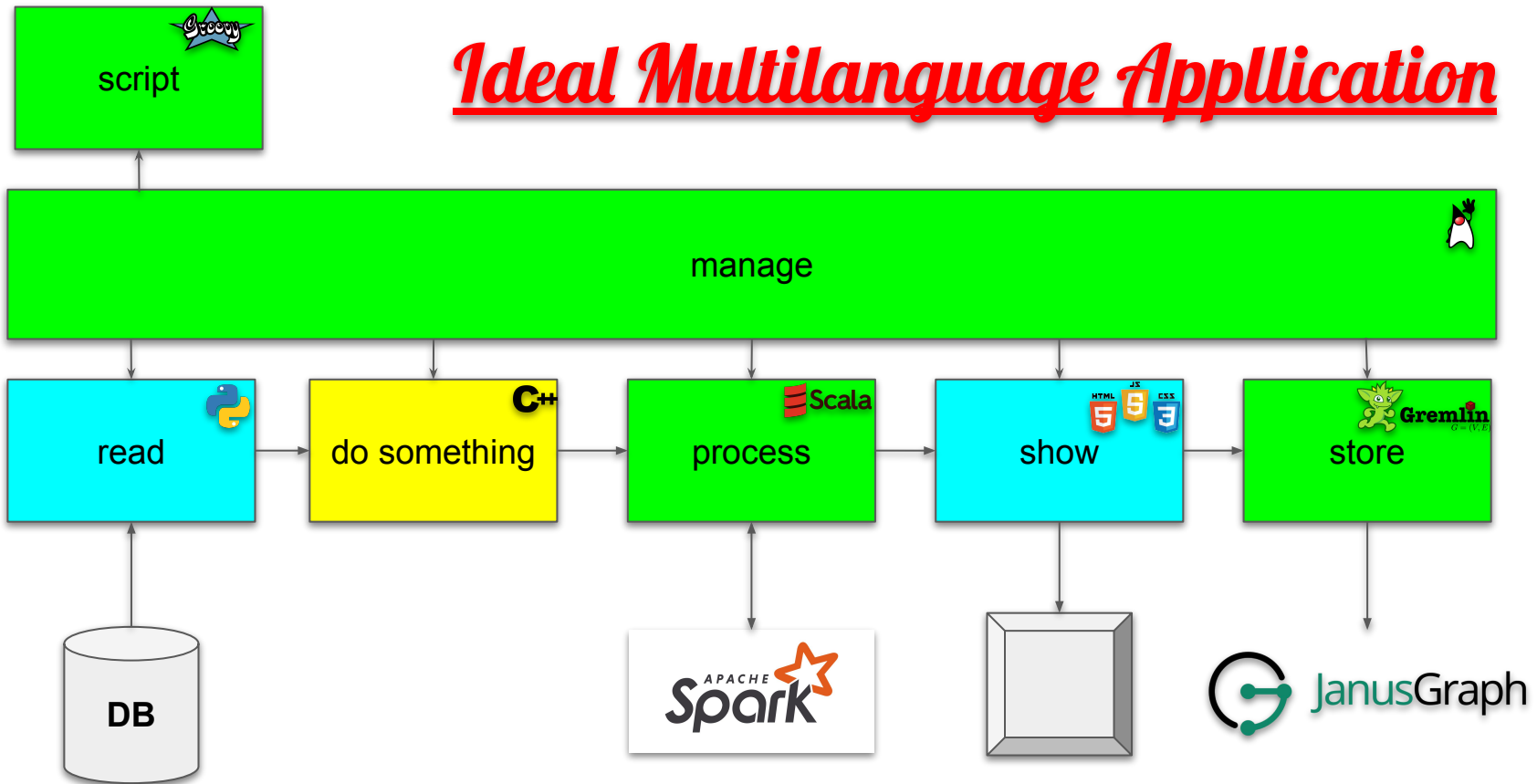
➤ Hybrid Storage advantages

- Expressiveness and flexibility of Graph Databases
- With performance and simplicity of tabular storage
- Under transparent interface



Multilanguage Environments

Ideal Multilanguage Application



Use the best tools and languages for each task.

Transparent interfaces (no stubs).

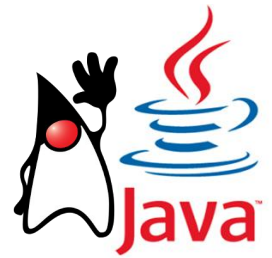
Data sharing (no proxies).

It works! - We are (almost) there!

***What is
the general multilanguage technology status
?***




JVM Languages



- Languages completely interoperable with **Java** (loaded into the same runtime or compiled into standard class-files)
- Fully inter-operable
- We can freely mix code from all those languages (even via inheritance)
- Can be used in a scripting interpreted way or compiled
- Successful new features from those languages are being incorporated in Java itself (e.g. functional syntax from Scala)

- **Groovy** (Apache): very high level scripting language, used in Graph DB
- **Scala** (Apache): functional language, used in Spark
- **Kotlin** (Google): for Android
- **Clojure**: Lisp-like
- **BeanShell**: interpreted/scripted Java



```
#!/usr/bin/env groovy
// converting SQL into XML with Groovy
// either run as a shell script or compiled
// -----
sql = Sql.newInstance("jdbc:mysql://localhost/Tuples",
                    "org.gjt.mm.mysql.Driver")
xml = new MarkupBuilder(new File("Tuples.xml"))
xml.tagSet() {
    sql.eachRow("select * from tuple where run > 2") {
        row -> xml.tag(Run:row.run, Event:row.event)
    }
}
```




Managed Languages



- Languages from different origin, made interoperable by re-implementation (or via specific bridges)
 - Go, Haskel, JavaScript, Lisp, OCaml, Pascal, PHP, Python, R, REXX, Ruby, Scheme, Smalltalk, Tcl,...
- More than 100 languages available in some way



C-World

- Direct compilation to native code
 - Sometimes by pre-compiling to C
- Lack of high level management (reflection, introspection)
 - Often implemented on top with in-house solutions
 - Which generates incompatibilities
- Often considered as faster and smaller
 - But even when it's true, there is a cost
 - Lack of functionality
 - Non-reproducibility
 - Non-portability
 - Very complex implementation of higher-level concepts
- Can be only connected via direct JNI or JNA
 - As they are running in an **unmanaged environment**
- Co-existence between managed JVM languages and low-level C-languages is difficult, proprietary or too primitive
 - No generic approach (so far)

***Revolution ?
(Holy Grail ?)***

*New Managed Environment
Supporting both JVM and C-based languages
To run in VM or natively*

➤ **Universal VM**

- Non-JVM languages are at the same level as JVM languages (=> full interoperability)
- All languages running in the same VM (traditional multi-language environment runs multiple languages side-by-side with frequent conversions of data)
- GraalVM is faster and smaller than OpenJVM (GraalVM is written in Java, OpenJVM is written in C++)
- Full interoperability between OpenJVM and GraalVM (program compiled for one can be run in another)
- Can be embedded in external applications (Oracle, Apache, MySQL,...)

➤ **Can build native executables and libraries** (using AOT (Ahead Of Time) compiler instead of JIT)

- Fully interoperable with native applications
- Smaller footprint, faster startup, sometimes faster execution
- Losing some dynamical features

- Polyglot (J)DK & (J)VM
- By Oracle
 - Big effort
 - Also included in OracleDB
 - Already used in industry (Twitter,...)
- CE (Community Edition): GPL licence (or less) - as Java
 - Components have the same licences as the original implementations (eg. Python as Python)
- EE (Enterprise Edition): better performance, security, support,...
- GraalVM JIT is included in OpenJDK (project *Galahad*):
`java -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler`
 - So trivial to try
 - Native Image compiler will follow
- New release every 3 months
 - rel22 supporting JDK 11,17
 - rel23-dev supporting JDK 17,20
- Linux, MS, MacOSX
- Uses new Java modularity features (since Java 9)
 - As the pluggable JIT compiler
- Similar project in the past: [NestedVM](#) - failed in 2009

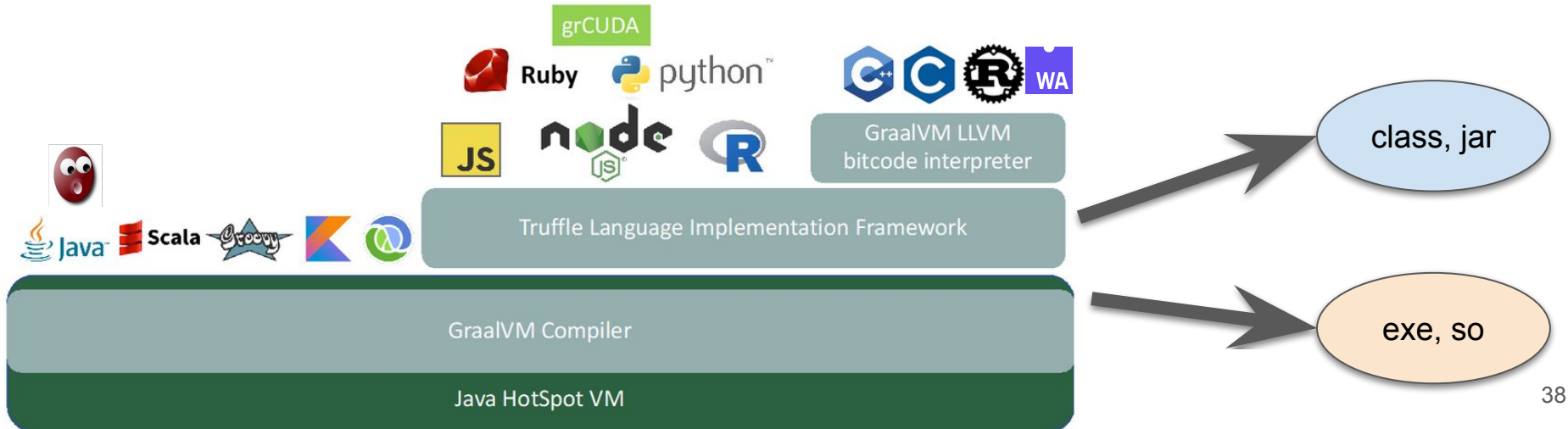
Supported Languages

- Growing number of supported languages (CUDA, WebAssembly,...)
- New Tools (debuggers, profilers, monitors,...)
- Integration in other applications and toolkits

Multiple languages are running in the same space/environment.

✗

Traditional multi-language pgms run multiple languages side-by-side.



Tools

- Growing number of supported languages (CUDA, WebAssembly,...)
- New Tools (debuggers, profilers, monitors....)
- Integration in other applications and toolkits

Tools understand your language.

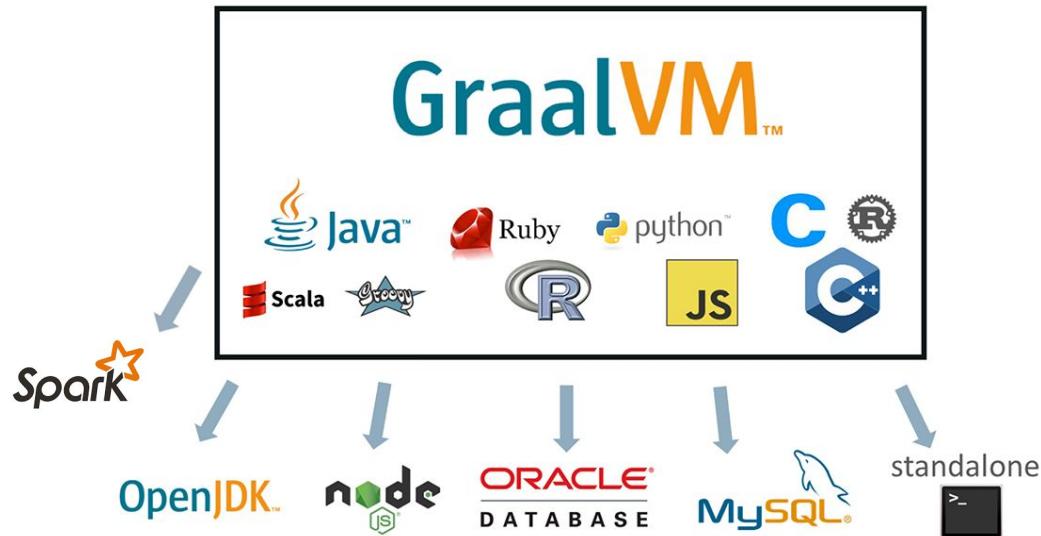
Unlike tools for pre-compiled languages.

Name	Count	Size	Retained
String	4,635 (0.3%)	444,960 B (0.5%)	n/a
Symbol	1,846 (0.1%)	177,216 B (0.2%)	n/a
Class	1,355 (0.1%)	130,080 B (0.1%)	n/a
Array	686 (0%)	65,856 B (0.1%)	n/a
Regexp	437 (0%)	41,952 B (0%)	n/a
Proc	379 (0%)	36,384 B (0%)	n/a
Proc#1695: lambda	96 B (0%)	96 B (0%)	n/a
Proc#1965: lambda	96 B (0%)	96 B (0%)	n/a
Proc#1926: block in define_hooked_variable	96 B (0%)	96 B (0%)	n/a
self (hidden): Module#365: Truffle:KernelOperations	96 B (0%)	96 B (0%)	n/a
block (hidden): null	-	-	-
references			
Proc#1967: lambda	96 B (0%)	96 B (0%)	n/a
Proc#2000: lambda	96 B (0%)	96 B (0%)	n/a
Proc#2001: block in define_hooked_variable	96 B (0%)	96 B (0%)	n/a
Proc#2002: lambda	96 B (0%)	96 B (0%)	n/a

Integration

- Growing number of supported languages (CUDA, WebAssembly,...)
- New Tools (debuggers, profilers, monitors,...)
- Integration in other applications and toolkits

*Allows, for example,
using MySQL with Python instead of SQL.*



Native Image Example

```
$ javac Hello.java
$ time java Hello
Hello !
0,10s user 0,03s system 131% cpu 0,097 total
$ native-image Hello
```

Basic Example

```
GraalVM Native Image: Generating 'hello'...
```

```
=====
[1/7] Initializing... (4.1s @ 0.21GB)
Version info: 'GraalVM 22.0.0.2 Java 11 CE'
[2/7] Performing analysis... [*****] (12.7s @ 0.47GB)
2,563 (82.60%) of 3,103 classes reachable
3,211 (60.36%) of 5,320 fields reachable
11,648 (72.43%) of 16,082 methods reachable
27 classes, 0 fields, and 135 methods registered for reflection
57 classes, 58 fields, and 51 methods registered for JNI access
[3/7] Building universe... (0.8s @ 0.62GB)
[4/7] Parsing methods... [*] (0.8s @ 0.84GB)
[5/7] Inlining methods... [****] (1.2s @ 0.75GB)
[6/7] Compiling methods... [****] (9.3s @ 1.19GB)
[7/7] Creating image... (1.1s @ 1.45GB)
3.69MB (35.06%) for code area: 6,949 compilation units
5.86MB (55.66%) for image heap: 1,543 classes and 80,509 objects
999.26KB (-9.28%) for other data
10.52MB in total
-----
Top 10 packages in code area:
606.25KB java.util
282.31KB java.lang
222.52KB java.util.regex
219.55KB java.text
193.17KB com.oracle.svm.jni
149.73KB java.util.concurrent
117.92KB java.math
103.60KB com.oracle.svm.core.reflect
97.83KB sun.text.normalizer
88.78KB com.oracle.svm.core.gencaveange
... 111 additional packages
(use GraalVM Dashboard to see all)
-----
1.6s (5.1% of total time) in 17 GCs | Peak RSS: 2.54GB | CPU load: 3.33
-----
```

```
Produced artifacts:
hello (executable)
hello.build_artifacts.txt
```

```
Finished generating 'hello' in 31.1s.
```

```
$ time hello
Hello !
0,00s user 0,00s system 89% cpu 0,002 total
```

```

${graalvm_dir}/bin/native-image \
--delay-class-initialization-to-runtime=\
io.grpc.netty.shaded.io.netty.handler.ssl.OpenSsl \
--initialize-at-build-time=\
org.apache.log4j.Level, \
org.apache.log4j.Layout, \
org.apache.log4j.PatternLayout, \
org.apache.log4j.Logger, \
org.apache.log4j.helpers.LogLoorg.apache.log4j.Level, \
org.apache.log4j.Priority, \
org.apache.log4j.LogManager, \
org.apache.log4j.helpers.Loader, \
org.apache.log4j.helpers.LogLog, \
org.apache.log4j.Category, \
org.apache.log4j.spi.RootLogger, \
org.apache.log4j.spi.LoggingEvent, \
org.slf4j.LoggerFactory, \
org.slf4j.impl.Log4jLoggerAdapter, \
org.slf4j.impl.StaticLoggerBinder, \
java.beans.Introspector, \
com.sun.beans.Introspector, \
com.sun.beans introspect.ClassInfo \
--report-unsupported-elements-at-runtime \
-H:Name=GroovyEL.exe \
-H:Path=./bin \
-jar ../lib/GroovyEL.exe.jar
```

Real-life Example

Polyglot Examples (1)

- Objects are never copied
- Conversion (into client physical format) at the latest possible time
- All tools are available for all languages
- **Several ways of calling foreign language:**
 - **Load as a script and execute**
 - **Compile as a class and use**
 - **Generate Native Image and call**

```
// Java calling C
Context context = Context.create();
File file = new File("polyglot"); // c-pgm compiled with GraalVM
Source source = Source.newBuilder("llvm", file).build();
Value cpart = polyglot.eval(source);
cpart.execute();
```

```
// Java calling Python
Value clazz = context.eval(Source.newBuilder("python", new File("mycode.py")).build());
Value instance = clazz.newInstance(1234);
System.out.println(instance.invokeMember("pyMethod", new int[]{1, 2, 3}));
```

```
// C calling JS
poly_create_context(thd, &ctx);
poly_context_eval(thd, ctx, "js", "foo", "function() {return 42;}", &func);
poly_value_execute(thd, func, NULL, 0, &answer);
poly_value_fits_in_int32(thd, answer, &fits);
poly_value_as_int32(thd, answer, &result);
return result;
```

```
// Java calling JS
Context context = Context.create();
Value v = context.eval("js", "function() {return 42;}");
Value answer = v.execute();
return answer.asInt();
```

Polyglot Examples (2)

- Interaction with LLVM languages requires more boiler-plate code
- It's simpler to compile JVM code into Native Image than to interface JVM with LLVM
- C++ calling Java is simpler than Java calling C++

// C++ calls Java

```
// C++
int main() {
    graal_isolate_t *isolate = NULL;
    graal_isolatethread_t *thread = NULL;
    graal_create_isolate(NULL, &isolate, &thread);
    printf("Result> %d\n", ceilingPowerOfTwo(thread, 14));
}

// Java
public class MyMath {
    @CEntryPoint (name = "ceilingPowerOfTwo")
    public static int ceilingPowerOfTwo(IsolateThread thread, int x) {
        return IntMath.ceilingPowerOfTwo(x);
    }
}
```

```
// JS calls CUDA
const DeviceArray = Polyglot.eval('grcuda', string='DeviceArray')
const in_arr = DeviceArray('float', 1000)
const out_arr = DeviceArray('float', 1000)
// set arrays ...
const code = '__global__ void inc_kernel(...) ...'
const buildkernel = Polyglot.eval('grcuda', string='buildkernel')
const incKernel = buildkernel(code, 'inc_kernel', 'pointer, pointer, uint64')
incKernel(160, 256)(out_arr, in_arr, N)
```

// JS calls C++

```
// JS
loadSource("llvm", "cpppart");
Value getSumOfArrayFn = polyglotCtx.getBindings("llvm").getMember("getSumOfArray");
int sum = getSumOfArrayFn.execute(sqrNumbers, sqrNumbers.length).asInt();

// C++
extern "C" getSumOfArray(int array[], int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += array[i];
    }
    return sum;
}
```

Where it is already useful **Now**

- Good news: **It really works and it works well**
- For JVM languages:
 - Just using GraalVM JIT (included in OpenJVM) makes it faster (better optimisation)
 - Compiling with GraalVM compiler make better bytecode
 - Creating Native Image may improve performance
 - Allows better integration with other languages
 - For Scala:
 - GraalVM JIT is able to optimize Scala much more than OpenJVM JIT (factor > 2)
- For Python:
 - Full interoperability with JVM languages
 - Speed, especially when compiled to Native Image
 - Better interoperability with C/C++ when compiled to Native Image
- For C/C++:
 - Can replace C/C++ code with code in better languages or integrate existing components written in better languages
 - By compiling them into Native Image or connecting with Truffle multi-language environment
 - Integration in frameworks written in other languages
 - Possibility to run in *Managed Environment* (so easy debugging)
 - Sometimes performance gain just by re-building using GraalVM (without modification)

***Can rewrite just one part of the system in another (more suitable) language,
And compile into native executable.***

Intrinsic Limitations

- It may be complicated to configure
 - In many cases, native image generation should be configured/tuned
 - One can/should configure/tune for performance
- Some (Java) applications may need JVM even when compiled into native executable
 - When they (mis)use reflection and construct classes at run-time
 - For example log4j
 - But after all, we may consider JVM just as another native library (which it is)
- We may gain speed for small applications, not so often for large complex ones
 - Not surprising, Java is often fast for real-life applications
- By compiling into native executable, we lose flexibility and portability
- Truffle languages (Python, Ruby, JS,...) are not at the same level of inter-operability as direct JVM languages
- Co-existence of LLVM languages (C, C++, Rust) with JVM languages is not as straightforward as between two JVM languages
 - Different memory & object models
 - Values, objects, names should be converted
 - Heavy communication across LLVM-JVM border may slow down execution
 - In that case, it may be more useful to compile JVM languages into native image
 - But it's probably as far as one can go in integrating JVM & C languages

External Complications

- Language specific build systems
 - Very elaborated make files
- Language specific deployment systems
 - Silently installing dependencies
 - Pip, conda, node, ...
- Specific bridges between languages
 - Often, internal implementation uses other languages
 - Python packages often contains C code, ...
- Language versions
 - It's impossible to support all language versions and dialects
 - Python 2 vs 3, ...
- Complex project specific environments

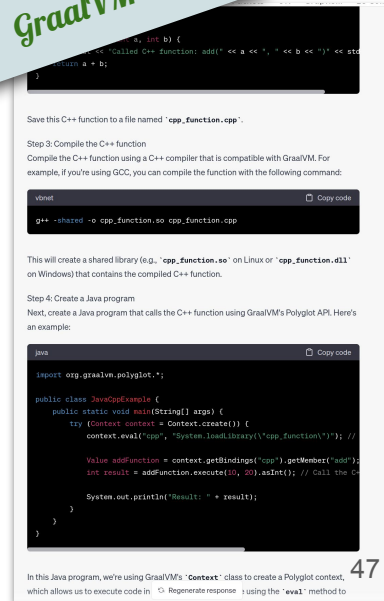
*Long list of projects which have already been ported/migrated/interfaced.
The most popular & least proprietary ones.*

Future of Programming

- The Frameworks will consist of various components ...
 - Third-party black-boxes
 - Written by AI
 - Legacy boxes
- Sometimes, we may not even know (or care) what is the implementation language
 - This already works in the classical JVM
- Languages will be used for their strong points (Scala for parallelism, JavaScript for Graphics,...)
 - We may stop creating proprietary languages (ADL, Root,...)
- Seamless (plug-in) ...
- It's important to really separate data from algorithms and logic (finally)

*Can rewrite just one part of the system in another (more suitable) language,
And compile into native executable.*

ChatGPT-generated
program using Java &
C++ connected by
GraalVM



```
int b) {
    // Called C++ function: add(1 << a << ", " << b << ") << std
    return a + b;
}

Save this C++ function to a file named 'cpp_function.cpp'.

Step 3: Compile the C++ function
Compile the C++ function using a C++ compiler that is compatible with GraalVM. For
example, if you're using GCC, you can compile the function with the following command:

$ g++ -shared -o cpp_function.so cpp_function.cpp

This will create a shared library (e.g., 'cpp_function.so' on Linux or 'cpp_function.dll'
on Windows) that contains the compiled C++ function.

Step 4: Create a Java program
Next, create a Java program that calls the C++ function using GraalVM's Polyglot API. Here's
an example:

$ java -cp org.graalvm.polyglot.*;

import org.graalvm.polyglot.*;

public class JavaCxxExample {
    public static void main(String[] args) {
        try {Context context = Context.create(); {
            context.eval("cpp", "System.loadLibrary('cpp_function');"); //
            Value addFunction = context.getBindings("cpp").getMember("add");
            int result = addFunction.execute(1, 2).asInt(); // Call the C
            System.out.println("Result: " + result);
        }
    }
}
```

In this Java program, we're using GraalVM's 'Context' class to create a Polyglot context, which allows us to execute code in `Regenerate response` using the 'eval' method to