МІНІСТЭРСТВА АДУКАЦЫІ РЭСПУБЛІКІ БЕЛАРУСЬ

БЕЛАРУСКІ ДЗЯРЖАЎНЫ ЎНІВЕРСІТЭТ

ФАКУЛЬТЭТ ПРЫКЛАДНОЙ МАТЭМАТЫКІ І ІНФАРМАТЫКІ

Кафедра тэхналогій праграмавання

ТАМАШЭВІЧ
Канстанцін Іванавіч

РАСПРАЦОЎКА РЭЛЯЦЫЙНАЙ БАЗЫ ДАДЗЕНЫХ, АПТЫМІЗАВАНАЙ ПАД ПАТРЭБЫ РАСПРАЦОЎКІ ГУЛЬНЯЎ

Дыпломная праца

Навуковы кіраўнік:

старэйшы выкладчык кафедры тэхналогій праграмавання

У.М. Гошка

Дапушчаны да абароны

«___» _____ 2021 г

Загадчык кафедры тэхналогій праграмавання

доктар тэхнічных навук, прафесар А. Н. Курбацкі.

Мінск 2021

ЗМЕСТ

Реферат

Дипломная работа, 82 ст., 17 рис., 7 источников.

Ключевые слова: РАЗРАБОТКА ИГР, C++, РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ, ШАБЛОНЫ ПРОЕКТИРОВАНИЯ ИГРОВЫХ СИМУЛЯЦИЙ, ШАБЛОН СУЩНОСТЬ-КОМПОНЕНТ-СИСТЕМА.

Объект исследования – шаблоны проектирования игровых симуляций, архитектура встраиваемых реляционных баз данных.

Цель работы – разработать встраиваемую базу данных для экспериментального использования в качестве системы сохранения и передачи данных в рамках шаблона Сущность-Компонент-Система.

Методы исследования – анализ, моделирование.

Результатом является встраиваемая реляционная база данных, которую можно использовать как экспериментальную систему сохранения и передачи данных в рамках шаблона Сущность-Компонент-Система.

Область применения – разработка игр и движков для разработки игр на C++.

Рэферат

Дыпломная праца, 82 ст., 17 мал., 7 крыніц.

Ключавыя словы: РАСПРАЦОЎКА ГУЛЬНЯЎ, C++, РЭЛЯЦЫЙНЫЯ БАЗЫ ДАДЗЕНЫХ, ШАБЛОНЫ ПРАЕКТАВАННЯ ГУЛЬНЁВЫХ СІМУЛЯЦЫЙ, ШАБЛОН ІСНАСЦЬ-КАМПАНЕНТ-СІСТЭМА.

Аб'ект даследвання – шаблоны праектавання гульнёвых сімуляцый, архітэктура ўбудаваных рэляцыйных баз дадзеных.

Мэта працы – распрацаваць ўбудаваную базу дадзеных для эксперыментальнага выкарыстання ў якасці сістэмы захавання і перадачы дадзеных у межах шаблона Існасць-Кампанент-Сістэма.

Метады даследвання – аналіз, мадэляванне.

Вынікам з'яўляецца ўбудаваная рэляцыйная база дадзеных, якую можна выкарыстоўваць як эксперыментальную сістэму захавання і перадачы дадзеных у межах шаблону Існасць-Кампанент-Сістэма.

Вобласць ужывання – распрацоўка гульняў і рухавікоў для распрацоўкі гульняў на C++.

ESSAY

Diploma, 82 p., 17 illustrations, 7 sources.

Keywords: GAME DEVELOPMENT, C++, RELATIONAL DATABASES, GAME SIMULATION DESIGN PATTERNS, ENTITY-COMPONENT-SYSTEM DESIGN PATTERN.

Objects of research are game simulation design patterns and embedded relational databases architecture.

Purpose is to develop embedded relational database for experimental usage as component storage and component provider for Entity-Component-System pattern.

Methods of research are analysis and modeling.

The result is embedded relational database, that can be used as experimental component storage and component provider in Entity-Component-System pattern.

Scope is C++ game development and game engine development.

УВОДЗІНЫ

Камп’ютарныя гульні з’яўляюцца дыскрэтнымі сімуляцыямі ў рэальным часе з адной ці больш асобамі, якія прымаюць рашэнні незалежна адна ад адной. Пры гэтым да кроку сімуляцыі прымяняюцца дастаткова жорсткія патрабаванні: у гульнях, выпушчаных раней за 2010 год, крок сімуляцыі павінен быў завяршацца за 34 мілісекунды ў 99% выпадкаў на адпавядаючай патрабаванням гульні сістэме. А ў сучасных гульнях патрабаванні павысіліся яшчэ больш: зараз крок павінен завяршацца за 17мс у 99.9% выпадкаў, калі сістэма адпавядае патрабаванням гульні. З-за такіх патрабаванняў звычайныя для класічных настольных і вэб-прыладаў методыкі і тэхналогіі не адпавядаюць патрабаванням для распрацоўкі гульняў, бо гэтыя тэхналогіі арыентаваны пад прадастаўленне вялікай колькасці карыстальнікаў доступу да вялікай колькасці дадзеных з больш мяккімі патрабаваннямі да часу выканання. Такім чынам, арыентацыя на прадастаўленне невялікай групе карыстальнікаў адносна невялікага аб’ёму дадзеных, але з жорскімі патрабаваннямі да хуткасці і стабільнасці часу выканання, задала іншы вектар развіцця тэхналогіям працы з дадзенымі ў сферы распрацоўкі гульняў.

Тым не менш, у апошнія гады ўсё больш папулярным становіцца аналіз і пошук паралеляў паміж шаблонамі праектавання ў распрацоўцы гульняў і ў звычайных бізнес-прыладах. На аснове гэтых даследванняў быў створаны шаблон праектавання Існасць-Кампанент-Сістэма, які ідэалагічна значна бліжэй да рэляцыйных структур, чым стандарт індустрыі – шаблон Сцэна-Вузел-Кампанент. Шаблон Існасць-Кампанент-Сістэма актыўна развіваецца і знаходзіць прымяненне ў індустрыі: на ім кампанія Blizzard распрацавала Overwatch, якая выіграла мноства прэмій і атрымала адзнакі 9/10 і 10/10 ад найбуйнейшых гульнёвых крытыкаў, а для вядомага гульнёвага рухавіка Unity быў распрацаваны эксперыментальны пакет Unity DOTS, арыентаваны пад распрацоўку гульняў па шаблону Існасць-Кампанент-Сістэма. Пры гэтым у шаблоне Існасць-Кампанент-Сістэма на дадзены момант няма адзінай спецыфікацыі алгарытму пошуку патрэбных існасцяў і перадачы іх сістэмам. У межах гэтай працы было вырашана паставіць эксперымент і напісаць убудаваную базу дадзеных, якая будзе выконваць задачы пошуку і перадачы існасцяў сістэмам.

Дыпломная праца складаецца з уводзін, трох глаў, заключэння і спіса літаратуры.

У першай главе разгледжаны найбольш выкарыстоўваемыя шаблоны праектавання гульнёвай логікі: ад шырока вядомых за межамі сферы

распрацоўкі гульняў да арыентаваных у першую чаргу на распрацоўку гульнёвай логікі. Пры гэтым увага звяртаецца на наяўнасць паралеляў паміж гэтымі шаблонамі і падобнымі механізмамі ў рэляцыйных базах дадзеных, што падкрэслівае магчымыя варыянты выкарыстання распрацоўваемай у гэтай працы эксперыментальнай базы дадзеных.

У другой главе разгледжаны фундаментальныя механізмы індэксацыі і алгарытмы выканання аперацыі злучэння ў рэляцыйных базах дадзеных, веданне якіх неабходна для рашэння найбольш прыярытэтнай для распрацоўваемай базы дадзеных задачы: эфектыўнай пастаўкі існасцяў сістэмам у шаблоне Існасць-Кампанент-Сістэма.

У трэцяй главе апісаны ўвесь працэс распрацоўкі эксперыментальнай базы дадзеных: ад пастаноўкі патрабаванняў да базы дадзеных да рэалізацыі канкрэтнага функцыянала і праверкі практычнай выкарыстоўваемасці распрацаванай базы дадзеных на простым эксперыментальным праекце.

ГЛАВА 1.
АСНОЎНЫЯ ШАБЛОНЫ ПРАЕКТАВАННЯ ГУЛЬНЁВАЙ ЛОГІКІ

У гэтай главе з дапамогай матэрыялаў з крыніц [1, с. 1013-1158] і [2, с. 19-104, с. 211-266] будуць разгледжаны асноўныя шаблоны праектавання, якія выкарыстоўваюцца пры праектаванні і рэалізацыі логікі гульнёвай сімуляцыі. Таксама з дапамогай матэрыялаў з крыніцы [5, с. 55-162] будуць праведзены паралелі паміж выкарыстоўваемымі ў гэтых шаблонах прынцыпамі і падобнымі да іх прынцыпамі баз дадзеных.

1.1. Базавыя шаблоны

У гэтым падзеле будуць разгледжаны шаблоны праектавання, аднолькава выкарыстоўваемыя як у сферы распрацоўкі гульняў, так і ў іншых сферах.

1.1.1 Індэксацыя

Як і ў многіх другіх прыладах, у гульнях часта ўзнікае неабходнасць кэшыравання вынікаў аперацый пошука для хуткага паўтарэння гэтых аперацый. Напрыклад, можа спатрэбіцца хутка знайсці ўсё аб’екты, якія падаюць у заданую прамавугольную вобласць ці перасякаюцца з дадзенай прамой. Структуры, якія дапамагаюць паскараць такія запыты, прынята называць кэшамі ці індэксамі. Часта спатрэбліваюцца наступныя тыпы індэксаў:

- Індэкс па поўнаму супадзенню параметра – на кожнае значэнне параметра вяртае масіў аб’ектаў, у якіх гэты параметр мае такое ж значэнне. З запытам такі індэкс павінны спраўляцца ці за амартызаваную канстанту O(1), ці за O(lgN), дзе N – колькасць усіх існуючых хаця б у адным аб’екце значэнняў параметра. Звычайна рэалізуецца хэш-картай, картай-дрэвам ці адсартыраваным масівам. Такія індэксы можна сустрэць у любым праекце, бо гэта самы просты і самы вядомы тып індэксаў.
- Індэкс па суадношанню значэнняў параметра – на кожны інтэрвал значэнняў параметра вяртае масіў аб’ектаў, падаючых у гэтых інтэрвал. Калі інтэрвал схлопаваецца да аднаго значэння, працуе як індэкс па поўнаму супадзенню параметра. З запытам такі індэкс павінны спраўляцца за O(lgN), дзе N – колькасць усіх існуючых хаця б у адным аб’екце значэнняў параметра. Звычайна гэты тып індэкса рэалізуецца картай-дрэвам ці адсартыраваным масівам. Такі тып індэксаў часта неабходны для рэалізацыі

складаных механік, напрыклад для рэалізацыі штучнага інтэлекту для персанажа-лекара, якому для прыняцца рашэння неабходна атрымаць спіс усіх саюзніказ з нізкім паказнікам здароўя.

- Індэкс па суадношанню значэнняў N-мернага картэжа параметраў – на кожную N-мерную фігуру, зададзеную праз інтэрвал значэнняў для кожнага вымярэння, і на кожны N-мерны прамень, зададзены ў параметрычнай форме, вяртае масіў аб'ектаў, усе выбраныя параметры якіх пападаюць у зададзеныя фігурай інтэрвалы ці усе выбраныя параметры якіх супадаюць з значэннямі прамяня. З запытам такі індэкс павінны спраўляцца за O(lgM), дзе M – колькасць унікальных картэжаў значэнняў параметраў аб'ектаў. Звычайна гэты тып рэалізуецца праз кропкавае N-мернае дрэва, якое з'яўляецца падвідам аб'ектных N-мерных дрэваў, якія будуць апісаны крыху пазней. Такі тып індэксаў часцей за ўсё выкарыстоўваецца для пошуку аб'ектаў у двумерных ці трохмерных сімуліруемых сусветаў па іх пазіцыях, напрыклад для пошуку ўсіх бачных на экране трохмерных аб'ектаў. Заўважым, што такі тып індэксаў выкарыстоўваецца з пазіцыямі аб'ектаў сусвету толькі тады, калі хуткасць атрымання прыярытэтней карэктнасці, бо звычайна памер аб'екта сусвету не роўны нулю і таму аб'ект не з'яўляецца кропкай.
- Індэкс па суадношанню значэнняў N-мернага картэжа параметраў, дзе значэнне кожнага параметра з'яўляецца інтэрвалам – на кожную N-мерную фігуру, зададзеную праз інтэрвал значэнняў для кожнага вымярэння, і на кожны N-мерны прамень, зададзены ў параметрычнай форме, вяртае масіў аб'ектаў, усе значэнні-інтэрвалы выбраных параметраў якіх перасякаюцца з зададзенымі фігурай інтэрваламі ці перасякаюцца з зададзеным прамянём. З запытам такі індэкс павінны спраўляцца за O(lgM), дзе M – колькасць унікальных картэжаў значэнняў-інтэрвалаў параметраў аб'ектаў. Такі складаны фармат неабходны для карэктнага пошуку аб'ектаў у сусвеце сімуляцыі, якія не з'яўляюцца кропкамі, а маюць ненулявы памер хаця б па адной асі. У двумерным варыянце такія аб'екты звычайна акругляюцца да абмяжоўваючага прамавугольніка, у трохмерным – да абмяжоўваючага прамавугольнага паралеліпіпеда, у N-мерным – абмяжоўваючай прамавугольнай N-мернай фігуры. Такое акругленне дазваляе задаваць памеры аб'ектаў як інтэрвалы па асях і з высокай акуратнасцю шукаць іх па гэтаму тыпу індэкса. Таму гэты індэкс традыцыйна выкарыстоўвацца для такіх алгарытмаў, як тапалагічная сартыроўка аб'ектаў у ізаметрычнай праекцыі, пошук бачных у камеры двумерных і трохмерных

аб’ектаў, трансляцыя кропкі кліка з экранных каардынат у каардынаты сусвету і гэтак далей.

Першыя тры тыпы індэксаў падтрымліваюцца большасцю сучасных баз дадзеных і шырока выкарыстоўвацца за межамі сферы распрацоўкі гульняў. Такім чынам, выкарыстанне ўбудаванай базы дадзеных дазволіла б не прапісваць індэксы ўручную у звязаных з німі структурах дадзеных, а выкарыстоўваць для гэтага метады ўбудаванай базы дадзеных. Чацвёрты варыянт выкарыстоўваецца толькі ддля спецыфічных задач пошуку па сусвету, але, пры магчымасці мадыфікацыі коду базы дадзеных, яго таксама можна перанесці на ўзровень базы дадзеных.

1.1.2 Рэактыўнае праграмаванне

Рэактыўнае праграмаванне – гэта парадыгма праграмавання, арыентаентавая на плыні дадзеных і распаўсюд зменаў. Гэта парадыгма дазваляе спрасціць выражэнне статычных і дынамічных плыняў дадзеных і гарантаваць тое, што змена адных элементаў мадэлі прывядзе да аўтаматычнай каскаднай змены залежных элементаў. Напрыклад, калі значэнне пераменнай A залежыць ад пераменных $B_1, B_2, \ldots, B_n$, то пры змене любой B_i значэнне A павінна быць пералічана.

Магчымасць імгненна рэагаваць на змены ў сістэме зрабіла парадыгму рэактыўнага праграмавання вельмі зручнай для мадэліравання сістэм, якія змяняюцца ў часе, і рэалізацыі карыстальніцкага інтэрфэйса. Гульны з’яўляюцца дыскрэтнымі сімуляцыямі ў рэальным часе, таму і ў ніх даволі часта выкарыстоўваецца гэта парадыгма. Найбольш частымі прыкладамі выкарыстання рэактыўнага праграмання ў гульнях з’яўляюцца:

- Апрацоўка падзей у фізічнай сімуляцыі. У найбольш папулярных зараз фізічных рухавікоў Bullet Physics і NVIDIA PhysX ёсць магчымасць дадання функтараў-прывязак на падзеі пачатку і сканчэнні сутыкнення паміж фізічнымі аб’ектамі, што дазваляе рэалізаваць рэактыўны падыход і імгненна рэагаваць на сутыкненні аб’ектаў.
- Аўтаматычная падтрымка карэктнасці залежных дадзеных у розных кампанентах. Даволі часта пры змене дадзеных у адным кампаненце неабходна абнавіць дадзеныя ва ўсіх звязаных кампанентах, напрыклад пры перамяшчэнні аб’екта абнавіць навігацыённую карту, параметры фізічнага цела аб’екта і паслаць паведамленне усім іерархічна залежным кампанентам пра неабходнасць такой жа аперацыі абнаўлення. Гэта можна рэалізоўваць і «у лоб», але выкарыстанне рэактыўнага

праграмавання дазваляе спрасціць даданне новых залежнасцяў і падтрымку кода праекта.

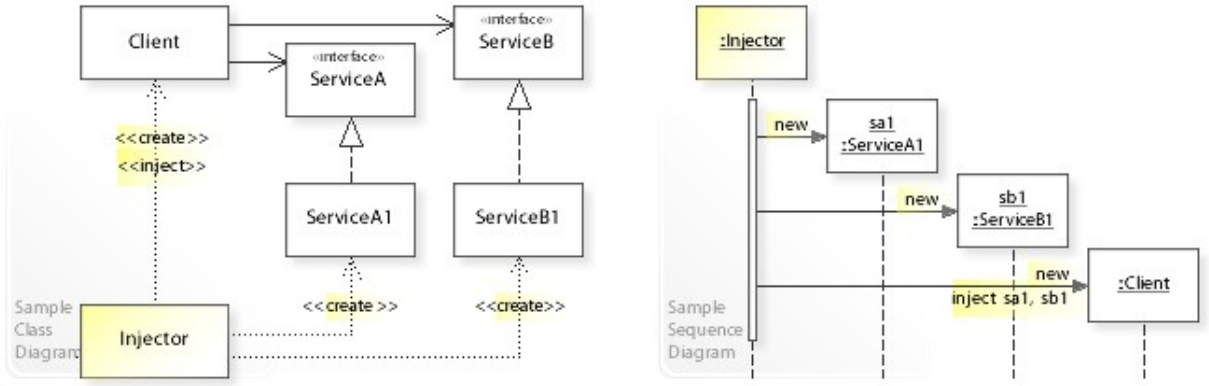
- Знешняя праверка карэктнасці праведзеных зменаў. Даволі часта прэдыкаты карэктнасці зменаў у гульнях дынамічныя і мяняюцца ў працэсе гульні, напрыклад персанаж можа патрапіць у павуцінне і часова згубіць магчымасць рухацца. Для такіх выпадкаў даданне знешніх прэдыкатаў праз сродкі рэактыўнага праграмання з'яўляецца дастаткова эфектыўным і гібкім рашэннем.

Чымсьці падобныя механізмы ў рэляцыйный базах дадзеных можна рэалізаваць праз прывязкі (англ. constraints) і трыгеры. Прывязкі дазваляюць дадаць прасцейшыя праверкі да значэнняў картэжаў. Трыгеры дазваляюць дазваляюць пісаць дастаткова складаныя функцыі праверкі і мадыфікацыя выконваемых з дадзенымі дзеянняў, тым самым рэалізуючы механізм прывязак (англ. callbacks) з рэактыўнага праграмавання. Такім чынам, выкарыстоўваемыя пры распрацоўкі гульняў прыёмы з рэактыўнага праграмавання можна пакрыць механізмамі баз дадзеных.

1.1.3 Уключэнне залежнасцяў

Уключэнне залежнасцяў – механізм, па якому адзін аб'ект атрымлівае ад нейкага пастаўшчыка іншыя аб'екты, ад якіх становіцца залежным. У працэсе ўключэння залежнасцяў існасці дзеляцца на наступныя тыпы:

- Кліенты – аб'екты, якія атрымліваюць залежнасці.
- Сэрвісы – аб'екты, якія ўключаюцца ў другія аб'екты як залежнасці.
- Уключальнікі – механізмы, якія на аснове нейкай логікі выбіраюць сэрвісы для кліентаў і ўключаюць іх як залежнасці кліентаў.



Выява 1.1 – UML дыяграммы, ілюструючыя ўключэнне залежнасцяў.

Механізм уключэння залежнасцяў пры правільным выкарыстанні дазваляе зменьшыць узаемасувязь розных аб’ектаў праз абстрагіраванне з дапамогай сэрвісаў. Існуе даволі многа класічных прыкладаў выкарыстання гэтага механізма пры распрацоўцы гульняў, прывядзем некаторыя з ніх:

- Фізічнае цела само па сабе ніколі не мае формы. У большасці сучасных гульнёвых рухавікоў, напрыклад Unity і Unreal Engine, у абгортках над фізічнымі рухавікамі існасць фізічнага цела поўнасцю аддзяляецца ад існасцяў фізічных формаў. Фізічныя формы аўтаматычна пастаўляюцца ў фізічные цела на этапе выканання гульні праз унутраныя механізмы уключэння залежнасцяў гульнёвых рухавікоў.
- Логіку прыняцця персанажам рашэнняў, напрыклад ісці ці атакаваць, звычайна аддзяляюць ад логікі выканання прынятых рашэнняў. Пры гэтым выконваючыя аб’екты звычайна з’яўляюцца кліентамі, а прымае рашэнні адзін абагульнены сэрвіс. Такім чынам для сістэм выканання рашэнняў не мае значэння, хто кіруе персанажам: гулец ці штучны інтэлект – у любым выпадку яны будуць атрымліваць каманды праз аднолькавы інтэрфейс.
- Пазіцыя існасці ў сусвеце сімуляцыі звычайна захоўваецца ў спецыяльным абагульненым сэрвіс. Гэта дазваляе пазіцыяніраваць абсалютна любыя тыпы аб’ектаў, якія потым атрымаюць сваю пазіцыю як сэрвіс: двумерныя малюнкі, трохмерныя мадэлі, гукі, зоны-трыгеры і таму падобнае.

Механізм уключэння залежнасцяў звычайна базіруецца на выкарыстанні інтэрфэйсаў і палімарфізма часу выканання, таму не мае відавочнага аналага ў рэляцыйных базах дадзеных. Але ёсць наступныя варыянты сумяшчэння яго з базамі дадзеных:

- Калі мы выкарыстоўваем убудаваную базу дадзеных, якая поўнасцю знаходзіцца ў памяці, і не плануем серыялізаваць яе, то можна ствараць табліцы прывязаных рэалізацый, якія будуць захоўваць два полі ў картэжы: указальнік на аб’ект-кліент і указальнік на аб’ект-сэрвіс. Але гэта даволі ненадзейны і неідэальны варыянт сувязі.
- Рэалізацыя абмену дадзенымі паміж сэрвісамі і кліентам праз спецыяльныя зарэзірваваныя картэжы, для якіх ствараюцца спецыяльныя табліцы. Гэта механізм з’яўляецца стандартам для шаблона Існасць-Кампанент-Сістэма, пра які будзе расказана ніжэй.

1.2. Шаблон Сцэна-Вузел-Кампанент

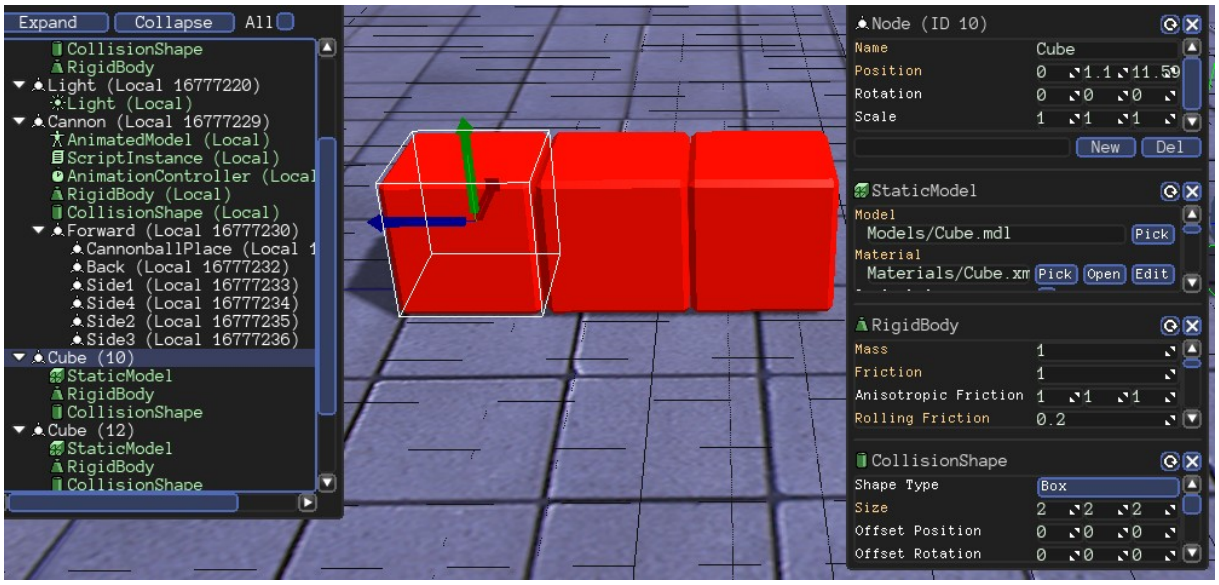
Шаблонам Сцэна-Вузел-Кампанент звычайна называюць цэлае сямейства шаблонаў праектавання гульнёвай логікі: Unity, Unreal Engine, Cry Engine і некаторыя іншыя гульнёвыя рухавікі маюць ідэйна падобныя стандартныя рэалізацыі структуры гульнёвай сімуляцыя, якія пры гэтым адрозньваюцца ў некаторых частках рэалізацый. Далей будзем разглядаць як шаблон Сцэна-Вузел-Кампанент рэалізацыю гэтых ідэй у двіжке з адчыненым ісходным кодам Urho3D. Пачнем разглядаць з пералічэння асноўных прынцыпаў:

- Структура сусвета сімуляцыі апісваецца як дрэва.
- Корань дрэва называецца сцэнай.
- Кожны вузел дрэва, уключаючы корань, можа захоўваць у сабе аб'екты. Такія аб'екты называюцца кампанентамі.
- Механізм уключэння залежнасцяў рэалізуецца праз даданне кампанентаў у вузлы.
- Кампаненты сцэны лічацца глабальнымі сэрвісамі і даступны ўсім кліентам.
- Кампанент можа выкарыстоўваць іншы кампанент як сэрвіс толькі тады, калі кампанент-сэрвіс знаходзіцца ў паддрэве, каранём якога з'яўляецца вузел з кампанентам-кліентам. Выключэнне з гэтага правіла – глабальныя сэрвісы.
- Кампаненты могуць абменьвацца паведамленнямі праз агульную шыну паведамленняў, доступ да якой прадстаўляе сцэна.

Вынясем асноўныя вызначэнні:

- Сцэна – корань дрэва сусвета сімуляцыі, які ўтрымлівае глабальныя сэрвісы, такія як сусвет фізічнай сімуляцыі і октадрэва, і прадстаўляе кампанентам доступ да шыны абмену паведамленнямі. У Urho3D сцэна дадаткова адказвае за сімуляцыю часу, сінхранізацыю сябе паміж удалённымі кліентамі і адпраўку рэсурсных пакетаў удалённым кліентам, калі неабходна.
- Вузел – вузел дрэва сусвета сімуляцыі, з’яўляецца кантэйнерам кампанентаў. У Urho3D дадаткова выконвае ролю сэрвіса пазіцыі ў сусвеце і дазваляе прывязваць да сабе любыя часовыя значэнні з унікальнымі імёнамі, тым самым рэалізуючы шаблон абстрактнага дакумента.
- Кампанент –аб’ект, што захоўваецца ў вузле, які можа адначасова з’яўляцца сэрвісам для іншых кампанентаў і кліентам, які залежыць ад іншых кампанентаў як ад сэрвісаў.

На малюнку ніжэй прадстаўлены скрыншот з рэдактара сцэн рухавіка Urho3D, з дапамогай якога лягчэй візуальна ўявіць структуру сімуляцыі пры выкарыстанні гэтага шаблона.



Выява 1.2 – Скрыншот з рэдактара сцэн рухавіка Urho3D.

У акне злева паказана структура сцэны, у якой белым напісаны імёны вузлоў, а зялёным – тыпы кампанентаў. Акно справа – рэдактар вузла, у якім можна рэдагаваць сам вузел і яго кампаненты.

На практыцы гэты шаблон апынуўся даволі простым і ўніверсальным, што зрабіла яго стандартам індустрыі на многія гады. Пералічым асноўныя яго плюсы:

- Трывіяльная і ўніверсальная рэалізацыя механізма ўключэння залежнасцяў.
- Структура дазваляе рэалізоўваць мнагаслойнае ўключэнне залежнасцяў без ускладнення коду.
- Пры правільнай рэалізацыі ўсе залежнасці, акрамя глабальных сэрвісаў, лакалізаваны ў паддрэве, што дазваляе выдзяляць як асобныя рэсурсы часткі сусвету, напрыклад дрэвы, дамы і гэтак далей. Такія выдзяленыя аб’екты звычайна называюць «загатоўкамі» (англ. prefab).
- Лёгкасць серыялізацыі сусвету як у тэкставыя, так і ў бінарныя фарматы.
- Магчымасць сур’ёзна змяняць існасць і паводзіны гульнёвых аб’ектаў без змянення коду за кошт змянення набораў кампанентаў у вузлах.
- Дрэвавідная іерархія і даданне ўласцівасцяў праз кампаненты на практыцы даволі лёгка тлумачыцца дызайнерам.

- Кампанентны падыход дазваляе атрымліваць спасылкі на сэрвісы дынамічна у той момант, калі яны становяцца неабходнымі. Пры гэтым прынята не захоўваць спасылкі на знойдзеныя сэрвісы, што дазваляе лёгка рэалізоўваць замену сэрвісаў «на ляту».

Але ёсць і некаторыя мінусы, якія падштурхоўваюць індустрыю да пошуку іншых шаблонаў праектавання структуры сімуляцыі:

- Дрэвавідная структура робіць больш складаным пошук аб’ектаў з неабходнымі ўласцівасцямі на сцэне, з-за чаго часта прыходзіцца траціць памяць і час выканання на стварэнне і падтрымку дадатковых індэксаў.
- Пошук адпаведных сэрвісаў сярод усіх кампанентаў алгарытмічна неаптымальны.
- Рэалізацыя ўключэння залежнасцяў праз інтэрфэйсы і віртуальныя метады не з’яўляецца дастаткова аптымальнай для высоканагружаных сімуляцый у некаторых гульнях.
- На вялікіх праектах становіцца даволі цяжка знаходзіць усе сувязі кліентаў з сэрвісамі, з-за таго што для большай гнуткасці мадэлі сэрвісы часта шукаюцца дынамічна і спасылкі на іх не кэшыруюцца.

З-за таго, што ў гэтым шаблоне шырока выкарыстоўваецца ўключэнне шаблонаў праз інтэрфэйсны падыход, яго даволі цяжка перанесці на механізмы рэляцыйных баз дадзеных: для атрымання сэрвісаў прыходзілася б ствараць дадатковыя адлюстраванні ці табліцы, тым самым павялічваючы выкарыстанне памяці гульнёй. Выкарыстанне памяці найбольш крытычна для гульняў, распрацоўваемых для мабільных прыладаў, бо ў іх, асабліва ў прыладах ад кампаніі Apple, часта працэсарная магутнасць больш прыярытэтны за аб’ём памяці. Напрыклад, да гэтага часу многія кампаніі, напрыклад Vizor Games, падтрымліваюць свае гульні на смартфоне iPhone 5s, у якога ўсяго адзін гігабайт памяці, што робіць аптымізацыю гульні пад яго даволі цяжкай задачай. Але, калі адкінуць падтрымку прыладаў з невялікай колькасцю памяці, выкарыстанне базы дадзеных для шаблона Сцэна-Вузел-Кампанент дазволіла б рашыць даволі важную праблему гэтага шаблона – складанасць пошуку аб’ектаў з-за дрэвавіднай структуры сімуляцыі.

1.3. Шаблон Існасць-Кампанент-Сістэма

Шаблон Сцэна-Вузел-Кампанент шырока выкарыстоўваецца і ў наш час, але апісаныя ў мінулым падзеле мінусы матывуюць інжынераў шукаць новыя варыянты арганізацыі структуры сусвету, якая б пры захаванні

большасць плюсаў шаблона Сцэна-Вузел-Кампанент мела б наступныя ўласцівасці:

- Больш наглядны механізм аб’яўлення і выкарыстання сэрвісаў, які б дазволіў спрасціць аналіз залежнасцяў у вялікіх праектах.
- Магчымасць гібкай рэалізацыі ўключэння залежнасцяў без выкарыстання інтэрфэйсаў і віртуальных метадаў.
- Лінейная ці блізкая да лінейнай структура сусвету, што дазваляе спрасціць рэалізацыю пошуку гульнёвых аб’ектаў.
- Магчымасць больш аптымальнага і арганізаванага працэса выдзялення памяці.

Шаблон Існасць-Кампанент-Сістэма (англ. Entity-Component-System) адказвае гэтым патрэбам. Першыя прататыпы гэтага шаблону распрацоўваліся яшчэ ў канцы 2000-х, але бурна набіраць папулярнасць ён пачаў толькі на працягу апошніх пяці год, пасля таго як ён быў выкарыстаны ў гульні Overwatch [3] і рэалізаваны ў папулярным рухавіку для распрацоўкі гульняў Unity [4]. Пачнем з пералічэння яго асноўных прынцыпаў:

- Логіка і дадзеныя павінны быць поўнасцю адлучаны адно ад аднаго.
- У абсалютнай большасці выпадкаў кампазіцыя лепей наследвання.
- Кампанент – набор дадзеных без канкрэтнай логікі іх апрацоўкі.
- Існасць – звязаны з унікальным ідэнтыфікатарам набор кампанентаў. Звычайна для спрашчэння пошуку памылак і працы дызайнераў таксама дадаецца магчымасць прысвайваць існасцям кароткія тэкставыя імёны.
- Сістэма – функцыя ці набор функцый, якія выклікаюцца кожны кадр і выконваюць праход па ўсіх існасцях, якія маюць кампаненты патрэбных фукнцыям тыпаў, чытаючы і мадыфікуючы іх, а таксама пры неабходнасці ствараючы новыя існасці і новыя кампаненты. Пры гэтым сістэмы не могуць выклікаць адна другую, але могуць выкарыстоўваць абагульненыя утылітарныя функцыі.
- Парадак выканання сістэмаў фіксіраваны і не можа змяняцца падчас выканання, але набор актыўных сістэм у агульным выпадку можа змяняцца падчас выканання сімуляцыі.

Заўважым, што гэтыя прынцыпы не даюць дакладнага адказу на наступныя пытанні:

- Ці можна выкарыстоўваць у шаблоне Існасць-Кампанент-Сістэма прынцыпы рэактыўнага праграмавання? Калі можна, то як?
- Як рэалізаваць механізм уключэння залежнасцяў, калі кампаненты не маюць логікі?
- Як арганізаваць эффектыўную падачу існасцяў сістэмам?

Пачнем з пытання прымянення рэактыўнага праграмавання ў шаблоне Існасць-Кампанент-Сістэма. Класічная схема выкарыстання рэактыўнага праграмавання праз прывязкі, праслухоўванне і знешнія верыфікатары лічыцца несумяшчальнай з «чыстым» варыянтам шаблона. Прычынай для гэтага з'яўляецца тое, што асноўная мэта шаблона Існасць-Кампанент-Сістэма – максімальна зменьшыць колькасць неяўных залежнасцяў паміж кампанентамі, а выкарыстанне прывязак прыводзіць як раз да зваротнага: змена нейкага значэння аднаго кампанента можа прывесці да каскаднай змены мноства іншых значэнняў іншых кампанентаў, што адразу супярэчыць і адсутнасці логікі ў кампанентах, бо яна неяўна дадаецца падчас выканання праз прывязкі, і патрабаванню яўнасці ўсіх зменаў, бо па аператару, які выконвае змену, нельга прадказаць, якія змены будуць выкананы разам з ім.

Але сама структура шаблона Існасць-Кампанент-Сістэма падказвае, што ў яе фармаце трэба глядзець на рэактыўнае праграмаванне з іншага боку. Кожны кадр сістэмы апрацоўваюць існасці і кампаненты, мяняючы іх, удаляючы старыя і ствараючы новыя. Такім чынам, кожная сістэма прынімае ў сабе плынь кампанентаў, апрацоўвае яе і выводзіць зменену плынь. А ўвесь набор сістэм фарміруе сабой модуль апрацоўкі плыні дадзеных, дзе на ўваход паступаюць старыя станы сімуляцыі, а выходзяць абноўленыя станы сімуляцыі. Нагадаем, што ідэяй рэактыўнага праграмавання з’яўляецца арганізацыя праграмы як генератара і апрацоўшчыка плыняў дадзеных. Такім чынам, негледзячы на несумяшчальнасць з класічнымі прыёмамі накшталт прывязак, шаблон Існасць-Кампанент-Сістэма ўтрымлівае прынцыпы рэактыўнага праграмавання як адну з сваіх базавых ідэй.

На практыцы перадача інфармацыі пра падзеі, напрыклад сутыкненні паміж аб’ектамі, вырашаецца праз спецыяльныя кампаненты. У Overwatch гэта рашаецца праз так званыя «адзінарныя кампаненты», якія з’яўляюцца ўнікальнымі ў межах сімуляцыі і звычайна замацоўваюцца на так званую «кантэкстную існасць». Усе сістэмы, якім трэба ведаць пра сутыкненні паміж аб’ектамі, выконваюцца пасля фізічнай сістэмы і чытаюць спіс сутыкненняў з адзінарнага кампанента фізічнай сістэмы. Заўважым, што такі механізм не дазваляе рэалізоўваць знешнюю праверку карэктнасці з адкатам зменаў, з-за чаго ў некаторых рухавіках ёсць рэалізацыі класічных

сродкаў рэактыўнага праграмавання. Але нават пры наяўнасці такіх сродкаў раіцца выкарыстоўваць іх толькі ў крытычных сітуацыях, бо некантраліруемае змешваенне класічнага рэактыўнага праграмавання з шаблонам Існасць-Кампанент-Сістэма звычайна прыводзіць да праблем пры дапрацоўцы кода.

Пяройдзем да рэалізацыі механізма ўключэння залежнасцяў. Як і з рэактыўным праграмаваннем, класічны падыход тут не працуе: уся логіка знаходзіцца ў сістэмах і іх утылітарных метадах, а значыць нельга ствараць інтэрфэйсы сэрвісаў і перадаваць праз кампаненты спасылкі на іх.

Але, як і з рэактыўным праграмаваннем, тут неабходна паглядзець на сам прынцып з другога боку. Уключэнне залежнасцяў падобна на так званую «чорную скрынку»: кліент ведае толькі інтэрфэйс – што трэба перадаць у « чорную скрынку », каб атрымаць неабходны набор дадзеных. Гэта дазваляе выразіць механізм наадварот, прадставіўшы кліента як пастаўшчыка дадзеных і прыёмніка вынікаў. Такая фармуліроўка трывіяльна рэалізуецца праз механізм шаблона Існасць-Кампанент-Сістэма: можна стварыць кампанент, у якім будуць палі-аргументы і палі-вынікі. Тады застанецца толькі сканфігураваць парадак выканання сістэмаў і іх функцый так, каб функцыя падрыхтоўкі аргументаў выканалася раней за ўсе магчымыя функцыі-рэалізацыі сэрвісаў, а функцыя апрацоўкі вынікаў заўсёды выконвалася пасля ўсіх рэалізацый функцый-сэрвісаў. А патрэбная функцыя-сэрвіс выбіраецца ў залежнасці ад набору кампанентаў існасці: звычайна кожнай рэалізацыі сэрвіса неабходна захоўваць свае дадатковыя дадзеныя, што дазваляе адназначна выбіраць функцыі-сэрвісы па наборах кампанентаў існасцяў.

Ідэя стварэння спецыяльнага кампанента для выкліку сэрвіса выглядае складанай і неаптымальнай. Але на практыцы рэдка даводзіцца выкарыстоўваць такое агульнае рашэнне, бо аргументамі сэрвіса амаль заўсёды з’яўляюцца ўжо існуючыя значэнні іншых кампанентаў існасці, таму неабходны толькі кампанент для вывада рашэнняў, прынятых сэрвісам. Можа адбыцца і наадварот – сэрвісу могуць быць патрэбны толькі ўваходныя дадзеныя, а яго вывадам з’яўляецца абнаўленне нейкіх спецыфічных для сэрвіса кампанентаў. Напрыклад, рэалізацыя штучнага інтэлекту сама можа забраць неабходныя дадзеныя з кампанентаў месцазнаходжання, здароўя, зброі і гэтак далей, пасля чаго яна вывядзе набор каманд у спецыяльны кампанент, які можа выкарыстоўвацца і для вываду каманд ад сапраўднага гульца, а не штучнага інтэлекту. Тады ўсе сістэмы, якім трэба прачытаць гэтыя рашэнні, змогуць прачытаць іх з гэтага кампанента і такім чынам самі стаць сэрвісамі па выкананню гэтых

рашэнняў, якія пры гэтым не будуць мець канкрэтнага абагульненага вывада.

Падача існасцяў у сістэмы адбываецца з дапамогай спецыяльных індэксаў. Адным з варыянтаў такіх індэксаў з'яўляецца захаванне для кожнага тыпу кампанента адсартыраванага па ідэнтыфікатарам масіваў існасцяў, у якіх ёсць гэты кампанент. Тады знайсці ўсе існасці, у якіх ёсць патрэбны набор кампанентаў, можна за $O(M*N)$, дзе M – колькасць тыпаў кампанентаў у наборы, а N – колькасць існасцяў у індэксу з найменшай колькасцю індэксаў сярод індэксаў для патрэбных тыпаў кампанентаў. Заўважым, што гэта з'яўляецца класічнай стратэгіяй для выканання аперацыі JOIN у рэляцыйных базах дадзеных, вядомай як стратэгія Merge Join.

Поўнае адзяленне дадзеных ад логікі ў шаблоне Існасць-Кампанент-Сістэма робіць адносна простым перанос некаторых яго механізмаў на механізмы рэляцыйных баз дадзеных. Пад кожны тып кампанента можна зрабіць табліцу і захоўваць кампаненты ў табліцах, звязваючы іх з існасцямі праз механізм знешніх ключоў. З-за таго што кампаненты не маюць логікі, перанос іх у базу дадзеных мала паўплывае на тое, як працуе гульня, але дасць наступныя плюсы:

- Пры запыце існасцяў з патрэбным наборам кампанентаў можна дадаваць фільтры па значэннях некаторых палёў кампанентаў, як пры звычайным SELECT-запыце з JOIN і WHERE. Такая функцыянальнасць не прысутнічае ў найбольш вядомых на дадзены момант рэалізацыях гэтага шаблона, але наяўнасць гэтай функцыянальнасці дазволіла б пазбавіцца ад так званых маркер-кампанентаў – кампанентаў, якія захоўваюць вельмі малы аб'ём дадзеных і асноўнай функцыяй якіх з'яўляецца не захаванне гэтых дадзеных, якія можна было б перанесці ў нейкі агульны кампанент, а маркіроўка існасці гэтым кампанентам для атрымання існасцяў з гэтым кампанентам ў канкрэтных сістэмах. Выдаленне маркер-кампанентаў дазволіць зменьшыць спажыванне памяці, бо не спатрэбіцца ствараць пад ніх алакатары, табліцы і індэксы.
- Тэорыя рэляцыйных баз дадзеных добра распрацавана і, дзякуючы падабенству кампанентаў да звычайных картэжаў дадзеных, дазволіць выкарыстоўваць праверанаыя часам фундаментальныя алгарытмы. Гэта дазволіць пазбегнуць марнавання часу на пераадкрыццё гэтых алгарытмаў і іх рэалізацыю канкрэтна пад шаблон Існасць-Кампанент-Сістэма.

- Выкарыстанне базы дадзеных рашае праблему серыялізацыі ўсёй сімуляцыі ці яе частак – гэта пяройдзе пад адказнасць базы дадзеных.

ВЫСНОВЫ

У гэтай главе былі разгледжаны найбольш выкарыстоўваемыя пры праектаванні логікі гульнёвай сімуляцыі шаблоны. Спачатку былі разгледжаны базавыя агульнавыкарыстоўваемыя падыходы, такія як:

- Індэксацыя, якая выкарыстоўваецца для паскарэння доступу да патрэбных элементаў.
- Рэактыўнае праграмаванне, якое выкарыстоўваецца для рэалізацыі лагічных сувязаў паміж рознымі аб'ектамі сімуляцыі.
- Уключэнне залежнасцяў, якое выкарыстоўваецца для павялічэння незалежнасці лагічных элементаў паміж сабой і дынамічнага стварэння аб’ектаў з рознымі ўласцівасцямі пад час выканання сімуляцыі.

Пасля гэтага былі апісаны два найбольш папулярных у сучаснай распрацоўцы гульняў комплексных падыхода да арганізацыі структуры і логікі сімуляцыі:

- Шаблон Сцэна-Вузел-Кампанент, які з'яўляецца найбольш шырока выкарыстоўваемым падыходам на працягу апошняга дзесяцігоддзя.
- Шаблон Існасць-Кампанент-Сістэма, які з'яўляецца пераасэнсаваннем шаблона Сцэна-Вузел-Кампанент і на працягу апошніх гадоў актыўна папулярызуецца як замена шаблону Сцэна-Вузел-Кампанент.

Пры апісанні і аналізу асобая ўвага была аддадзена пошуку магчымасцяў пераносу апісваемых шаблонаў на фундаментальныя механізмы рэляцыйных баз дадзеных. У выніку аналізу было вырашана, што шаблон Існасць-Кампанент-Сістэма добра карэліруе з механізмамі рэляцыйных баз дадзеных і можа быць перанесены на іх, пры гэтым стаўшы яшчэ больш універсальным і зручным для выкарыстання.

ГЛАВА 2.
ІНДЭКСАЦЫЯ І АПЕРАЦЫЯ ЗЛУЧЭННЯ Ў РЭЛЯЦЫЙНЫХ БАЗАХ ДАДЗЕНЫХ

Гульнёвы код, які пабудаваны з выкарыстанем шаблона Існасць-Кампанент-Сістэма, базіруецца вакол аперацыі перадачы сістэмам існасцяў, кампаненты якіх адпавядаюць нейкаму набору крытэрыяў, напрыклад маюць патрэбныя тыпы, як гэта зроблена ў Unity DOTS [4]. Нагадаем, што сістэма – гэта модуль апрацоўкі, які чытае і мадыфікуе зададзеныя на этапе кампіляцыі тыпы кампанентаў ў кожнай атрыманай на ўваход існасці, а таксама не мае свайго ўнутранага стану. Такім чынам, перадачу існасцяў сістэме можна выразіць праз механізмы рэляцыйных баз дадзеных як аперацыю ўнутранага злучэння кампанентаў па ўнікальным ідэнтыфікатарам іх існасцяў з фільтрацыяй па дадатковых крытэрыях. Для паскарэння гэтай аперацыі звычайна неабходна ствараць індэксы па выкарыстоўваемых палях, напрыклад на палі з унікальнымі ідэнтыфікатарамі існасцяў. Па гэтай прычыне ў дадзенай главе з дапамогай матэрыялаў з крыніц [6] і [7] будзе разгледжана індэксацыя ў рэляцыйных базах дадзеных і алгарытмы выканання аперацыі злучэння.

2.1. Індэксацыя

2.1.1 Мэты індэксацыі

Адным з галоўных патрабаванняў да баз дадзеных з’яўляецца магчымасць хуткага выканання аперацый пошуку для часта выкарыстоўваемых крытэрыяў. Пад хуткім выкананнем разумеецца тое, што алгарытм пошуку патрэбных элементаў павінен мець меншую асімптатычную складанасць па часу за лінейны пошук. Традыцыйна ў базах дадзеных гэта задача рашаецца праз індэксацыю: стварэнне дадатковых структур для паскарэння пошуку ці арганізацыю саміх дадзеных у больш зручным для выканання пошуку фармаце. А структура, якая паскарае выкананне пошуку ці апісвае мадыфікаваны фармат захавання дадзеных, завецца індэксам. Такім чынам, праз стварэнне індэксаў адміністратары базы дадзеных могуць паскараць выкананне тых аперацый пошука і змены дадзеных, якія неабходна часта выконваць пры працы ў абслугоўваемай прадметнай вобласці.

Але індэксацыя неабходна не толькі для паскарэнне запытаў абслугоўваемай прадметнай вобласці, але і для падтрымкі ўнутраных механізмаў, такіх як:

- Першасныя ключы.

- Знешнія ключы.
- Палі з ўнікальнымі значэннямі.

Звычайна базы дадзеных аўтаматычна ствараюць індэксы для звязаных з гэтымі механізмамі палёў, такім чынам гарантуючы, што база дадзеных аптымальна працуе з гэтымі механізмамі.

Для аптымальнага выканання запытаў розных прадметных сфераў распрацавана мноства тыпаў індэксаў. Напрыклад, Microsoft SQL Server 2019 падтрымлівае 12 тыпаў індэксаў, што пакрывае ўсе часта з'яўляючыяся ў індустрыі праблемы. Але ў гэтым раздзеле будуць разгледжаны толькі асноўныя фундаментальныя тыпы індэксаў.

2.1.2 Кластэрызуючыя індэксы

Кластэрызуючы індэкс – гэта індэкс, які задае фармат файла, у якім захоўваюцца дадзеныя табліцы. Такім чынам, у любой табліцы можа існаваць не больш аднаго кластэрызуючага індэкса, інакш узнікаў бы канфлікт пры вызначэнні фармату захавання запісаў у табліцы. Кластэрызуючыя індэксы з'яўляюцца эфектыўным рашэннем для дасягнення наступных мэтаў:

- Хуткае выкананне запытаў, аперыруючых на наборы запісаў, якія пры сартыроўцы ўсёй табліцы па нейкаму падмноству палёў фарміруюць адзін ці некалькі непарыўных блокаў. У гэтым выпадку ствараецца кластэрызуючы індэкс па выбранаму падмноству палёў, які арганізуе ўсе фізічныя прадстаўленні дадзеных у файле ў парадку, які задаецца сартыроўкай па выбранаму падмноству палёў. Правільны выбар кластэрызуючых індэксаў дазваляе значна павялічыць хуткасць выканання запытаў дзякуючы павялічэнню колькасці пападанняў у кэш, што ў разы змяншае колькасць запытаў на чытанне дадзеных з самаго файла на знешнім сховішчы. Разгледзім наступны выпадак: ёсць табліца, у якой захоўваюцца транзакцыі кожнага дэпартаменту фірмы. Неабходна з максімальнай хуткасцю падлічваць баланс любога падмноства дэпартаментаў за дадзены прамежак часу. Для гэтага неабходна стварыць кластэрызуючы індэкс па палях нумар_дэпартамента і дата_транзакцыі. Пры гэтым нумар_дэпартамента выкарыстоўваецца як першае прыярытэтнае поле для сартыроўкі з-за таго што лічыцца, што колькасць транзакцый заўсёды ў разы больш за колькасць дэпартаментаў. Такім чынам, пры запыце баланса для кожнага дэпартамента з нейкага падмноства за дадзены прамежак часу апрацоўваемыя картэжы сфарміруюць столькі непрырыўных блокаў, сколькі было

дэпартаментаў у падмностве, а апрацоўка кожнага блока займе мінімальна магчымы час дзякуючы іх непарыўнасці ў фізічным прадстаўленні.

- Змяньшэнне аб’ёму памяці на знешнім сховішчы, які займаюць дадзеныя. Некаторыя базы дадзеных з дапамогай кластэрызуючага індэкса могуць пры групіроўцы выносіць агульныя значэнні індэксіруемых палёў, такім чынам запіс часта выкарыстоўваемага значэння на знешняе сховішча адбываецца толькі адзін раз, а не паўтараецца для кожнага запісу. Гэта робіць фармат файлу з дадзенымі больш складаным, але ў некаторых выпадках дазваляе адчувальна зменьшыць аб’ём займаемай памяці. Разгледзім наступны выпадак: у кампаніі ёсць некалькі дэпартаментаў, а ў кожным дэпартаменце каля ста супрацоўнікаў, пры гэтым у табліцы з інфармацыяй пра супрацоўнікаў ёсць поле нумар_дэпартамента, у якім кожны супрацоўнік звязваецца з дэпартаментам. Стварэнне кластэрызуючага індэкса для гэтага поля дазволіць не толькі паскорыць усе запыты, якія апрацоўваюць усіх супрацоўнікаў канкрэтнага дэпартамента, але і зменьшыць выкарыстанне памяці, бо значэння поля нумар_дэпартамента будзе захоўвацца столькі раз, сколькі існуе дэпартаментаў, а не столькі раз, сколькі існуе супрацоўнікаў, а супрацоўнікаў у выпадку, які мы разглядаем, у 100 разоў больш.

Такім чынам, кластэрызуючыя індэксы з’яўляюцца моцным інструментам для аптымізацыі як хуткасці выканання нейкага падмноства запытаў, так і аб’ёму памяці, займаемага дадзенымі. Але аптымізацыя дасягаецца за кошт змены фармата файла дадзеных, што робіць немагчымым існаванне некалькіх кластэрызуючых індэксаў у адной табліцы. У многіх выпадках немагчыма пакрыць усе запыты, якія неабходна аптымізаваць, аднім кластэрызуючым індэксам, з-за чаго з’явіліся так званыя некластэрызуючыя індэксы. Таксама заўважым, што калі палі, па якіх пабудаваны кластэрызуючы індэкс, часта змяняюцца, то выкарыстанне кластэрызуючага індэксу можа наадварот зменьшыць хуткасць працы запытаў з табліцай, што таксама абмяжоўвае магчымасці выкарыстання гэтага тыпу індэксаў.

2.1.3 Упарадкаваныя некластэрызуючыя індэксы

Як абмяркоўвалася вышэй, у некаторых выпадках кластэрызуючыя індэксы не могуць пакрыць усе выпадкі выкарыстання табліцы ці наогул не ствараюцца з-за дынамічнасці змянення запісаў у прадметнай вобласці. Тады для паскарэння выканання запытаў выкарыстоўваюцца ўпарадкаваныя некластэрызуючыя індэксы. Упарадкаваны

некластэрызуючы індэкс з’яўляецца упарадкаваным наборам пар, дзе першае значэнне пары – набор значэнняў індэксіруемых палёў для нейкага запісу, а другое значэнне – указальнік на гэты запіс.

Калі пары ў індэксы ствараюцца для кожнага радка, то такі індэкс называюць шчыльным. Шчыльныя індэксы займаюць адносна вялікі аб’ём памяці, але дазваляюць вельмі хутка выконваць аперацыі пошуку.

Выява 2.1 – Візуалізацыя сувязі паміж шчыльным індэксам па назве краіны (злева) і табліцай з інфармацыяй пра краіны (зправа).

Праблемай шчыльных індэксаў з'яўляецца не толькі аб'ём памяці, які яны займаюць, але і выдаткі на падрымку карэктнасці гэтых індэксаў пры даданні, змене ці выдаленні запісаў. У тых выпадках, калі гэтыя выдаткі становяцца значнымі, выкарыстоўваюцца разрэжданыя індэксы, у якіх захоўваюцца пары толькі для часткі запісаў, якія разам з іншымі праігнарыраванымі запісамі фарміруюць блокі запісаў. Тады пры пошуку запісаў з дапамогай індэкса выбіраюцца адпаведныя блокі, пошук унутры якіх адбываецца лінейна.

Выява 2.2 – Візуалізацыя сувязі паміж разрэджаным індэксам па назве краіны (злева) і табліцай з інфармацыяй пра краіны (зправа).

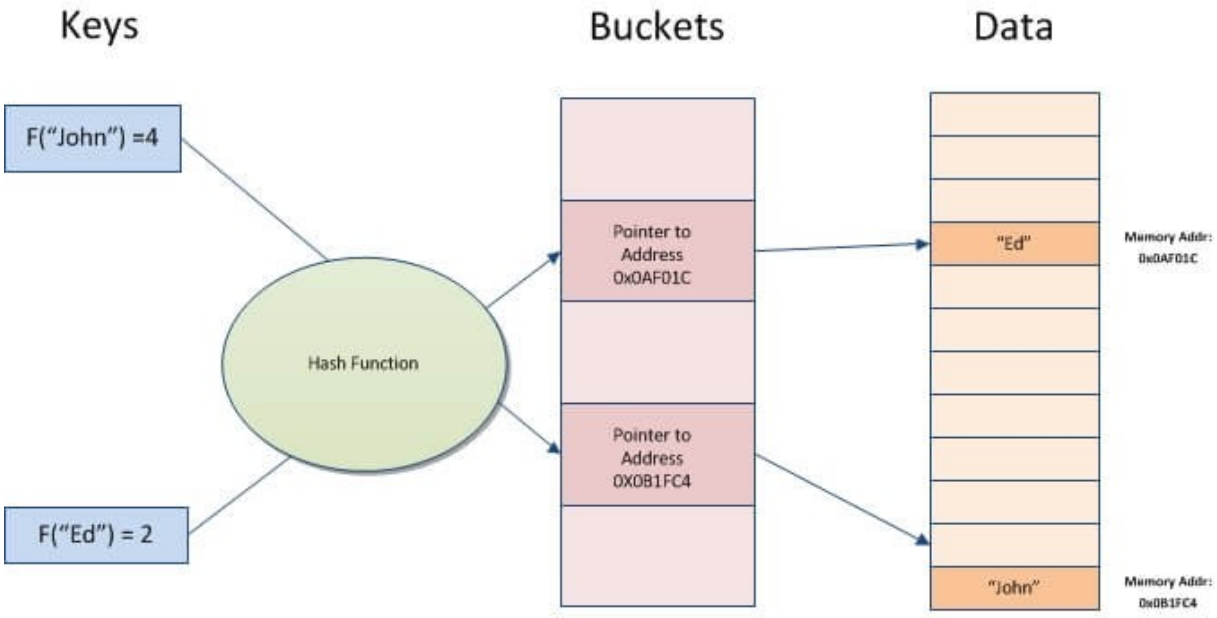
Такім чынам упарадкаваныя некластэрызуючыя індэксы дазваляюць паскараць патрэбныя тыпы запытаў не дыктуючы фармат захавання запісаў у файле дадзеных.

2.1.4 Хэш індэксы

Як кластэрызуючыя індэксы, так і ўпарадкаваныя некластэрызуючыя індэксы выкарыстоўваюць ідэю сартыроўкі па значэннях індэксіруемых палёў і выкарыстання бінарнага пошуку па атрыманаму адсартыраванаму масіву. Гэта дазваляе паскорыць як пошук канкрэтных запісаў, так і пошук

інтэрвалаў. Але бывае так, што запытам прадметнай вобласці неабходна знаходзіць не інтэрвалы запісаў, а адзін канкрэтны запіс, пры гэтым выконваючы такі пошук як мага хутчэй. У такім выпадку лагарыфмічная складанасць па часу бінарнага пошука здаецца ўжо не самым лепшым варыянтам.

Выкарыстанне хэш табліц дазваляе выконваць пошук канкрэтных запісаў са складанасцю па часу роўнай амартызаванай канстанце. Гэта значна хутчэй за бінарны пошук, але робіць немагчымым пошук інтэрвалаў запісаў. Тым не менш, пошук канкрэтнага запісу з'яўляецца настолькі частым выпадкам, што для рашэння гэтай задачы быў выдзелены асобны тып індэксаў – хэш індэксы.



Выява 2.3 – Візуалізацыя сувязі паміж хэш індэксам па полю з імем (злева) і табліцай з інфармацыяй (справа).

Як структурна, так і алгарытмічна хэш індэксы нагадваюць звычайныя хэш карты. Да значэнняў індэксіруемых палёў прымяняецца хэш функцыя, якая вяртае для кожнай групы значэння нумар так званага вядра, у якім захоўваецца ўказальнік на запіс ці групу запісаў, для значэнняў якіх хэш фукнцыя вярнула такі нумар. Як і ў хэш картах, пры наяўнасці калізіі пошук адбываецца сярод усіх запісаў, звязаных з атрыманым вядром.

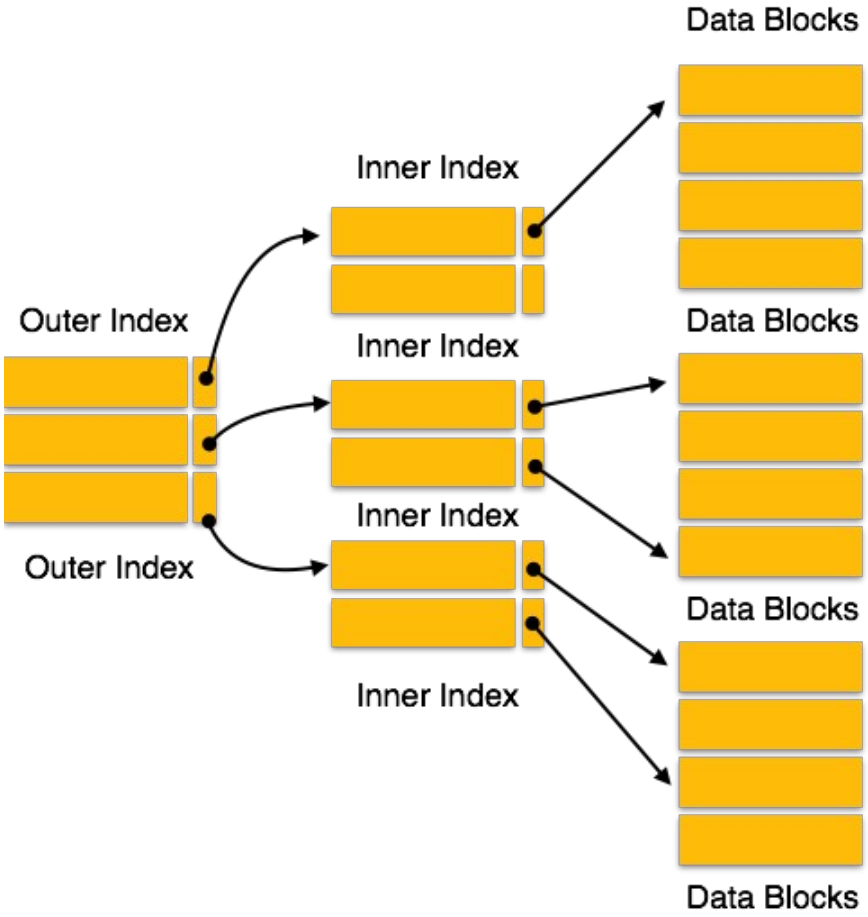
Такім чынам, хэш індэксы з'яўляюцца эфектыўным механізмам для паскарэння пошуку канкрэтных запісаў, а не інтэрвалаў. Пры гэтым хэш індэкс звычайна займае меншы аб'ём памяці, чым аналагічны

ўпарадкаваны некластэрызуючы індэкс. Гэта адбываецца дзякуючы таму што хэш індэкс не дубліруе значэнні індэксіруемых палёў.

2.1.5 Многаўзроўневыя індэксы

У табліцах з вялікай колькасцю запісаў упарадкаваныя некластэрызуючыя індэксы, асабліва плотныя, перастаюць змяшчацца ў аператыўную памяць, што прыводзіць да неабходнасці падгрузкі інфармацыі са знешняга сховішча і адчувальна зніжае хуткасць працы запытаў з такімі індэксамі. Для рашэння гэтай праблемы выкарыстоўваюцца так званыя многаўзроўневыя індэксы.

Самым простым варыянтам многаўзроўневага індэкса з’яўляецца двухслойны індэкс, які складаецца са знешняга вузла і набора ўнутраных вузлоў. Знешні вузел дазваляе па дадзенаму набору значэнняў у індэксіруемых палях выбраць адзін адпаведны ўнутраны вузел, а ўнутраныя вузлы ў сваю чаргу з’яўляюцца ўпарадкаванымі некластэрызуючымі індэксамі, пабудаванымі для падмноства запісаў табліцы. Пры гэтым усе вузлы будуюцца такім чынам, што кожны вузел памяшчаецца ў аператыўную памяць. Такім чынам для пошуку любога запісу па значэннях у яго індэксіраваных палях заўсёды неабходна толькі адна аперацыя чытання з знешняга сховішча, якая вяртае выбраны ўнутраны вузел. Калі б пошук выконваўся па звычайнаму ўпарадкаванаму некластэрызуючаму індэксу, які не памяшчаецца ў памяць, то колькасць такіх аперацый была б адчувальна большай.



Выява 2.4 – Візуалізацыя структуры двухслойнага індэкса.

Але ў некатарых выпадках у табліцы так многа запісаў, што немагчыма выканаць падзяленне на вузлы такім чынам, каб кожны вузел памяшчаўся ў аператыўную памяць. У такім выпадку замест двухслойнага індэксу будуецца збалансаванае дрэва пошуку, якое часта называюць B^{+}-дрэвам. B^{+}-дрэвы структурна нагадваюць двухслойныя індэксы і рэалізуюць падобны механізм на мностве слаёў, але ў гэтай працы B^{+}-дрэвы падрабязна разбірацца не будуць.

2.2. Алгарытмы выканання аперацыі злучэння

Аперацыя злучэння з рэляцыйнай алгебры дазваляе аб’ядноўваць запісы з двух і больш табліц па нейкай умове. Гэта аперацыя дазваляе рашаць такія задачы, як будова суадносінаў супрацоўнікаў з іх дэпартаментамі, дэпартаментаў з іх кіраўнікамі і гэтак далей.

Мэтай гэтай працы з’яўляецца будова базы дадзеных, якая будзе выкарыстоўвацца як сховішча кампанентаў для шаблона Існасць-Кампанент-Сістэма. Найбольш важнай задачай гэта базы дадзеных з’яўляецца эфектыўная перадача існасцяў, якія з’яўляюцца наборамі кампанентаў, сістэмам на апрацоўку. Гэту задачу можна

перафармуліраваць як выкананне умоўнага злучэння на некалькіх табліцах, дзе кожная табліца захоўвае кампаненты канкрэтных тыпаў і ўнікальныя ідэнтыфікатары існасцяў, да якіх яны прыналежаць. Такіх чынам перадаваемая сістэме існасць будзе кампазітным запісам, сфарміраваным як вынік злучэння кампанентаў з аднолькавым значэнням унікальнага ідэнтыфікатара існасці.

У агульным выпадку злучэнне з'яўляецца даволі складанай аперацыяй, але для часта адбываючыхся сцэнароў былі распрацаваны аптымізаваныя алгарытмы. У гэтым падзеле будуць разгледжаны асноўныя алгарытмы выканання злучэння, якія падтрымліваюцца ў большасці сучасных рэляцыйных баз дадзеных. Алгарытмы будуць разглядацца для злучэння двух табліц, таму што, як не цяжка даказаць праз матэматычную індукцыю, злучэнне трох і больш табліц заўсёды можна выразіць праз некалькі аперацый злучэння двух табліц.

2.2.1 Злучэнне з укладзеным цыклам

Злучэнне з укладзеным цыклам з'яўляецца самым трывіяльным алгарытмам злучэння і для двух табліц складаецца з наступных крокаў:

- Адной з табліц прысвойваецца статус знешняй, а другой – статус унутранай. Калі ў табліц няма індэксаў на выкарыстоўваемыя ва ўмове палі, то табліцца з меншай колькасцю запісаў будзе знешняй, бо гэта дазволіць больш эфектыўна выканаць алгарытм.
- Паслядоўна сканіруюцца ўсе запісы знешняй табліцы і для кожнага запісу выконваецца паслядоўнае сканіраванне запісаў унутранай табліцы, вынікам якога будуць запісы ўнутранай табліцы, якія можна злучыць з абраным запісам знешняй табліцы.

Гэты алгарытм з'яўляецца даволі неэфектыўным і займае вельмі шмат часу ў тых выпадках, калі ў абедзвюх табліцах вельмі многа запісаў. Але гэта адзіны алгарытм, які можна выкарыстоўваць для тых выпадкаў, калі прэдыкат злучэння зададзены не праз аперацыю ¿. Алгарытмы, якія мы будзем разглядаць далей, у большасці выпадкаў працуюць значна хутчэй, але патрабуюць злучэння праз прэдыкат з аперацыяй ¿.

Заўважым, што хуткасць працы злучэння з укладзеным цыклам можна падняць праз даданне індэкса на выкарыстоўваемыя прэдыкатам палі ва ўнутранай табліцы. Такая аптымізацыя з'яўляецца тыповым прымяненнем прынцыпа балансіравання часу-памяці, бо даданне індэкса павялічвае выкарыстанне памяці. Але пры наяўнасці адпаведнага індэкса ці пры выкананні на невялікіх табліцах злучэнне праз укладзены цыкл паказвае даволі добрыя вынікі і знаходзіць выкарыстанне ў тых выпадках, калі трэба апрацоўваць вялікую колькасць невялікіх транзакцый.

2.2.2 Злучэнне з хэшыраваннем

Як было сказана раней, калі прэдыкат злучэння выкарыстоўвае аператар ¿, абедзве табліцы маюць вялікі памер і ва ўнутранай табліцы адсутнічае індэкс па выкарыстоўваемых прэдыкатам палях, злучэнне з укладзеным цыклам паказвае не вельмі добрыя вынікі. У такіх выпадках часта выкарыстоўваецца злучэнне з хэшыраваннем, якое для дзвюх табліц складаецца з наступных крокаў:

- Па наступных правілах выбіраецца хэшыруемая табліца:
  - Калі хэш карта для кожнай табліцы умесціцца ў аператыўную памяць, то выбіраецца табліца з большай колькасцю запісаў.
  - Калі хэш карта толькі для адной табліцы памесціцца ў аператыўную памяць, то выбіраецца гэта табліца.
  - Калі хэш карты для любой табліцы не памесцяцца ў памяць, то выбіраецца табліца з меншай колькасцю запісаў. Заўважым, што часцей за ўсё ў гэтым выпадку сучасныя базы дадзеных выбіраюць іншы алгарытм злучэння.
- Для хэшыруемай табліцы будуецца хэш карта па выкарыстоўваемых у прэдыкаце з аперацыяй ¿ палях.
- Выконваецца паслядоўны праход па ўсіх запісах другой табліцы, дзе для кожнага запісу шукаюцца пары для злучэння ў хэшыруемай табліцы з дапамогай хэш карты.

Стварэнне хэш карты для табліцы гэта не вельмі танная аперацыя, але на практыцы злучэнне з хэшыраваннем заўсёды хутчэй непаскоранага індэксамі злучэння з укладзеным цыклам на адносна вялікіх табліцах. Таму большасць сучасных рэалізацый аперацыі злучэння выкарыстоўвае злучэнне з хэшыраваннем у тых выпадках, калі абедзве табліцы не маюць індэкса па выкарыстоўваемых у прэдыкаце палях і захоўваюць дастаткова вялікую колькасць запісаў, але хэш карта для хаця б адной з гэтых табліц памяшчаецца ў аператыўную памяць.

Можна заўважыць, што калі пры выкарыстанні злучэння з укладзеным цыклам ва ўнутранай табліцы ёсць хэш індэкс па выкарыстоўваемых у прэдыкаце з аперацыяй ¿ палях, то алгарытм злучэння звярнецца да злучэння з хэшыраваннем без будовы хэш карты. Але злучэнне з хэшыраваннем прынята выдзяляць у асобную катэгорыю не з-за выкарыстання хэш карты, а з-за таго, што гэта хэш карта будуецца пры выкананні запыта, а значыць не займае месца на знешнім сховішчы і не запаволвае выкананне мадыфікуючых табліцу запытаў, у адрозненні ад хэш індэкса.

2.2.3 Зліццё

Калі ў абедзвюх табліцах існуе індэкс, які можа прадставіць запісы як спіс, адсартыраваны па выкарыстоўваемых у прэдыкаце з аперацыяй ¿ палях, то злучэнне можна выканаць праз адзін цыкл, які працуе адначасова з запісамі абедзвюх табліц. На гэтым факце пабудаванны алгарытм зліцця, які для двух табліц складаецца з наступных крокаў:

- Калі ў першай табліцы адсутнічае ўпарадкаваны індэкс па выкарыстоўваемых у прэдыкаце з аперацыяй ¿ палях, то будуецца часовы адсартыраваны спіс запісаў, які будзе выкарыстоўвацца для ітэрацыі. Інакш для ітэрацыі будзе выкарыстоўвацца напрамую курсор па адпаведнаму індэксу.
- Выконваецца аналагічная праверка для другой табліцы і будуецца адсартыраваны спіс запісаў для яе, калі гэта неабходна.
- З дапамогай атрыманых ітэратараў адбываецца праход і злучэнне запісаў табліц. Напрыклад, у тым выпадку, калі ў табліцах наборы значэнняў у выкарыстаных для сартыроўкі палях не дубліруюцца і запісы адсартыраваны ў парадку ўзрастання, то цела цыкла трывіяльна:
    - Калі адзін з ітэратараў указвае на канец свайго спіса, цыкл перарываецца.
    - Выконваецца параўнанне запісаў, на якія ўказваюць ітэратары, па выкарыстаных для сартыроўкі палях.
    - Калі першы запіс меншы, пераводзім першы ітэратар на наступны запіс.
    - Калі другі запіс меншы, пераводзім другі ітэратар на наступны запіс.
    - Калі запісы роўныя, то вылічваем значэнне ўсяго прэдыката і ў выпадку поспеху злучаем запісы. Інакш пераводзім абодва ітэратары на наступныя запісы.

Калі ў абедзвюх табліцах прысутнічаюць упарадкаваныя індэксы па палях, выкарыстоўваемых у прэдыкаце з аператарам ¿, зліццё з’яўляецца аптымальным алгарытмам для выканання злучэння. Таксама зліццё часта выкарыстоўваецца замест злучэння з хэшыраваннем у тых выпадках, калі хэш карта для абаіх табліц не памесціцца ў аператыўную памяць, бо нават пры адсутнасці ўпарадкаваных індэксаў зліццё паказвае сабе лепей за злучэнне з хэшыраваннем на вельмі вялікіх табліцах.

ВЫСНОВЫ

У гэтай главе былі разгледжаны два найбольш важных для рэалізацыі эксперыментальнай базы дадзеных, арыентаванай пад працу з шаблонам Існасць-Кампанент-Сістэма, механізма рэляцыйных баз дадзеных: індэксацыя і алгарытмы выканання злучэння запісаў.

Пры гэтым былі разгледжаны фундаментальныя тыпы індэксаў у рэляцыйных базах дадзеных:

- Кластэрызуючыя індэксы, якія дазваляюць аптымізаваць саму структуру захавання дадзеных пад выконваемыя задачы.
- Шчыльныя і разрэджаныя ўпарадкаваныя некластэрызуючыя індэксы, якія дазваляюць аптымізаваць выкананне патрэбных запытаў без змены структуры захавання дадзеных.
- Хэш індэксы, які з'яўляюцца найбольш эфектыўным варыянтам для тых выпадкаў, калі запыты выконваюць аперацыю з адным канкрэтным запісам, а не інтэрвалам запісаў.
- Многаўзроўневыя індэксы, якія з'яўляюцца эфектыўнай рэалізацыяй ўпарадкаваных некластэрызуючых індэксаў для табліц з вялікай колькасцю запісаў.

Таксама былі растлумачаны фундаментальныя алгарытмы выканання аперацыі злучэння:

- Злучэнне з ўкладзеным цыклам як базавы ўніверсальны алгарытм.
- Злучэнне з хэшыраваннем як найбольш эфектыўны алгарытм пры адсутнасці індэксаў па патрэбных палях.
- Зліццё як найбольш эфектыўны алгарытм пры наяўнасці ўпарадкаваных індэксаў па патрэбных палях.

Такім чынам, у гэтай главе былі разгледжаны ўсе фундаментальныя механізмы, неабходныя для рэалізацыі задачы эфектыўнай перадачы існасцяў сістэмам у межах шаблона Існасць-Кампанент-Сістэма.

ГЛАВА 3.
РАСПРАЦОЎКА ЎБУДАВАНАЙ БАЗЫ ДАДЗЕНЫХ ДЛЯ ВЫКАРЫСТАННЯ З ШАБЛОНАМ ІКС

3.1. Патрабаванні да распрацоўваемай базы дадзеных

У першай главе былі разгледжаны і прааналізаваны асноўныя шаблоны праектавання структуры і логікі гульнёвых сімуляцый. Пры гэтым быў зроблены вывад, што многія з іх можна дастаткова эфектыўна пабудаваць паўзверх існуючых механізмаў рэляцыйных баз дадзеных. Але ў індустрыі базы дадзеных выкарыстоўваюцца толькі для захавання і счытвання дадзеных і не ўдзельнічаюць у саміх сімуляцыях. Асноўнымі прычынамі гэта з’яўляюцца:

- Выкарыстанне базы дадзеных як знешняга сэрвіса, напрыклад лакальна запушчанага сервера PostgreSQL, прыводзіць да накладных выдаткаў на перадачу дадзеных і затрымак пры перадачы дадзеных, з-за чаго сімуляцыя перастае ўкладвацца ў патрабаванні па затрачваемаму на падлік аднаго крока часу. Такім чынам, выкарыстоўваемая база дадзеных заўсёды павінна быць убудаванай у праграму гульні.
- Накладныя выдаткі на капіраванне картэжаў дадзеных пры іх выбары з табліц з'яўляюцца залішне вялікімі для сімуляцый большасці гульняў.
- У класічных рэляцыйных базах дадзеных адсутнічае магчымасць так званага «звароту па слабой спасылцы», які дазваляе вельмі хутка атрымаць картэж без капіравання ці даведацца пра тое, што гэты картэж больш не існуе. Аналагам гэтага ў імператыўных мовах праграмавання з'яўляецца зварот па ўказальніку ці спасылцы, які неабходны для эфектыўнай рэалізацыі многіх алгарытмаў.
- Па апісаных вышэй прычынах многія утылітарныя бібліятэкі, такія як рухавікі фізічных сімуляцый, напісаны без аглядкі на магчымасць выкарыстання з убудаванымі базамі дадзеных і звяртаюцца да сваіх дадзеных па спасылках, што робіць немагчымым іх выкарыстанне без накладных расходаў на капіраванне дадзеных з базы і назад у базу. Шырокае выкарыстанне гэтых бібліятэк само па сабе становіцца новай прычынай не выкарыстоўваць убудаваныя рэляцыйныя базы дадзеных.

У гэтай працы была пастаўлена мэта правесціі эксперымент і паспрабаваць напісаць на C++ убудаваную рэляцыйную базу дадзеных, пры выкарыстанні якой не ўзнікалі б апісаныя вышэй праблемы і якую такім чынам можна было эфектыўна выкарыстоўваць пры распрацоўцы гульняў. Да гэтай рэалізацыі базы дадзеных былі пабудаваны наступныя агульныя патрабаванні:

- Яна павінна быць убудаванай базай дадзеных.
- Яна павінна падтрымліваць наступныя стандартныя механізмы рэляцыйных баз дадзеных:
  - Стварэнне і выдаленне табліц падчас выканання.
  - SELECT-запыты ці аналагічны ім механізм.
  - UPDATE-запыты ці аналагічны ім механізм.
  - INSERT-запыты ці аналагічны ім механізм.
  - DELETE-запыты ці аналагічны ім механізм.
  - Спецыфікатар WHERE для SELECT, UPDATE і DELETE запытаў ці аналагічны яму механізм.
  - Выкарыстанне JOIN у SELECT і UPDATE запытах ці аналагічны яму механізм.
  - Сістэма індэксаў, якія можна ствараць і выдаляць падчас выканання, ці аналагічны ім механізм.
- Захоўваемыя ў табліцах картэжы павінны быць экземплярамі структур ці класаў, каб з імі можна было без залішніх цяжкасцяў працаваць у кодзе, не арыентаваным на працу з базамі дадзеных.
- База дадзеных павінна падтрымліваць даданне падчас выканання прывязак ці трыгераў, напісаных на C++.
- Вынікі запытаў павінны перадавацца не праз капіраванне, а праз указальнікі.
- Калі ў карыстальніцкага коду ёсць карэктны ўказальнік на картэж, то яму не трэба выконваць поўны SELECT ці UPDATE запыт, а дастаткова толькі паведаміць базу дадзеных пра пачатак і канец працы з аб’ектам.

Асноўнай задачай, якую павінна рашаць распрацоўваемая эксперыментальная база дадзеных, з’яўляецца перадача існасцяў сістэмам для рэалізацыі шаблона Існасць-Кампанент-Сістэма. Нагадаем, што ў гэтым шаблоне сістэма з’яўляецца апрацоўшчыкам, які не мае свайго ўнутранага стану, кампанент – структура дадзеных, якая не мае ніякай логікі, акрамя ўтылітарных метадаў, а існасць – набор кампанентаў, праз які выражаны нейкі аб’ект сімуліруемага сусвету гульні.

Разгледзім гэту задачу больш падрабязна. Пры кожным кроку сімуляцыі адбываецца выклік усіх сістэм, якія чакаюць перадачы

патрэбных ім існасцяў для выканання аперацый над імі. У залежнасці ад складанасці сімуляцыі колькасць сістэм ў ёй змяняецца ад 10 у простых казуальных гульнях да 200 у гульнях з мноствам складаных механік. Пры гэтым адзін крок павінен выконвацца за 17 мілісекунд у большасці выпадкаў, калі мы арыентуемся пад сучасныя патрабаванні: 60 крокаў за 1 секунду [3]. Пры гэтым чакаецца, што большасць адведзенага часу будзе выдзелена на выкананне аперацый над існасцямі, а не на пошук існасцяў для сістэм.

Запыт на існасці ад сістэмы звычайна складаецца з набора пакампанентных патрабаванняў: сістэма ўказвае, якія кампаненты ёй патрэбны для чытання і якія для запісу, а таксама можа дадаваць патрабаванні да значэнняў канкрэтных палёў кампанентаў. Пры гэтым для аптымізацыі звычайна патрабуецца, каб запыт быў канстантай. Адказам на такі запыт з'яўляецца плынь, якая, у залежнасці ад рэалізацыі, прадстаўляе сістэме ці спасылкі на існасці, у якіх ёсць усе запатрабаваныя кампаненты, якія адпавядаюць перададзеным крытэрыям, ці самі наборы запатрабаваных кампанентаў, якія знаходзяцца ў адной існасці. Першы падыход звычайна больш просты ў рэалізацыі, а другі падыход дазваляе гарантаваць тое, што сістэма будзе выкарыстоўваць толькі запытаныя кампаненты і не будзе спрабаваць звяртацца да іншых кампанентаў існасці. У дадзенай рэалізацыі было вырашана прытрымлівацца другога падыходу.

Пасля разбору задачы не цяжка заўважыць, што пошук і перадачу існасцяў сістэмам можна рэалізаваць як выкананне аперацыі злучэння запісаў з рэляцыйных баз дадзеных. На аснове праведзенага разбору задачы вызначым, якія алгарытмы выканання злучэння, разгледжаныя ў другой главе, падыходзяць для рашэння задачы:

- Злучэнне з ўкладзеным цыклам з'яўляецца недастаткова эфектыўным для рашэння гэтай задачы, але гэты алгарытм неабходна рэалізаваць як запасную стратэгію для тых выпадкаў, калі ў табліцах кампанентаў адсутнічае адпаведны індэкс. Таксама гэты алгарытм можна выкарыстоўваць тады, калі колькасць кампанентаў хаця б аднаго з патрэбных тыпаў не больш за 10 адзінак, тады хуткасць выканання аперацыі будзе дастатковай для рашэння задачы выбара існасцяў з такім рэдкім тыпам кампанентаў.
- Злучэнне з хэшыраваннем з'яўляецца добрай стратэгіяй для тых выпадкаў, калі аперацыя злучэння выконваецца не вельмі часта і таму не мае сэнсу ствараць для яе індэкс. Але ў разглядаемай задачы за адну секунду выконваецца некалькі соцень ці нават тысяч такіх аперацый, таму стварэнне хэш карты для кожнай

аперацыі з’яўляецца неэфектыўным рашэннем, а значыць і такі алгарытм злучэння зусім не падыходзіць для гэтай задачы.

- Зліццё з’яўляецца аптымальным алгарытмам для рашаемай задачы, бо злучэнне па ўнікальнаму ідэнтыфікатару гарантуе наяўнасць аперацыі ¿ у прэдыкаце і магчымасць сартыроўкі па палях, звязаных з гэтай аперацыяй. Пры гэтым зліццё з’яўляецца найбольш эфектыўнай стратэгіяй пры наяўнасці патрэбных індэксаў у табліцах кампанентаў, а дадаваць такія індэксы пры выкарыстанні шаблона Існасць-Кампанент-Сістэма можна нават аўтаматычна, бо запыты сістэм канстантны і пры ініцыялізацыі сусвету заўсёды можна выкарыстаць гэты факт для дадання патрэбных індэксаў.

Пасля вызначэння адпаведных алгарытмаў злучэння разгледзім і выбярам тыпы індэксаў, якія дапамогуць у аптымізацыі выканання злучэння:

- Кластэрызуючыя індэксы не могуць быць рэалізаваны ў распрацоўваемай базе дадзеных з-за таго што яны змяняюць фармат, у якім захоўваюцца, у тым ліку ў памяці, запісы табліц. А змяняць гэты фармат у час выканання нельга з-за таго, што ад распрацоўваемай базы дадзеных патрабуецца, каб запісы былі структурамі ці класамі, што дазваляе зручна працаваць з імі ў тых частках коду, якія павінны быць абстрагіраваны ад выкарыстання базы дадзеных.
- Упарадкаваныя некластэрызуючыя індэксы, асабліва іх шчыльны варыянт, неабходны для эфектыўнай рэалізацыя злучэння праз зліццё, таму павінен прысутнічаць у распрацоўваемай базе дадзеных.
- Хэш індэксы могуць дапамагчы аптымізаваць запыты, злучэнне ў якіх праводзіцца праз алгарытм злучэння з укладзеным цыклам. Але, калі неабходна аптымізаваць запыт праз даданне індэкса, то больш лагічна дадаць упарадкаваны некластэрызуючы індэкс, каб злучэнне можна было выконваць адразу праз зліццё, што больш эфектыўна ў межах рашаемай задачы. Таму было вырашана не дадаваць хэш індэксы ў эксперыментальную базу дадзеных.
- Многаўзроўневыя індэксы з’яўляюцца эфектыўным рашэннем для тых выпадкаў, калі ўпарадкаваны некластэрызуючы індэкс не памяшчаецца ў аператыўную памяць. Але ўся сімуляцыі, а значыць і ўсе табліцы базы дадзеных, павінны памяшчацца ў аператыўную памяць, інакш будзе немагчыма выконваць крокі сімуляцыі з патрэбнай хуткасцю. Таму ніколі не можа ўзнікнуць патрэба у індэксе, які не памяшчаецца ў аператыўную памяць.

Такім чынам рэалізоўваць многаўзроўневыя індэксы для эксперыментальнай базы дадзеных няма сэнсу.

Вызначыўшы задачу, патрэбныя алгарытмы злучэння і тыпы індэксаў, сфармуліруем канчатковыя патрабаванні да рэалізацыі індэксацыі і злучэння ў рэалізоўваемай эксперыментальнай базе дадзеных:

- Павінна быць магчымасць ствараць шчыльныя ўпарадкаваныя некластэрызуючыя індэксы ў любой табліцы па любых палях.
- Карэктнасць індэксаў у табліцах павінна падтрымлівацца аўтаматычна.
- Павінна быць магчымасць ствараць таблічныя плыні чытання і рэдагавання, вяртаючыя запісы ў зададзеным абраным індэксам парадку.
- Павінен існаваць фармат задання запыту на злучэнне запісаў з розных табліц і алгарытм аўтаматычнага выбару выкарыстоўваемых індэксаў і алгарытма злучэння для гэтага запыту.
- Павінны быць рэалізаваны алгарытмы злучэння праз зліццё і праз злучэнне з укладзеным цыклам, якія выконваюць злучэнне па дадзенаму ў спецыяльным фармаце запыце.

3.2. Прывязкі і транзакцыённая мадэль памяці

Да прывязак, якія выконваюць праверку значэнняў палёў пры ўстаўцы і абнаўленні картэжаў і адкат гэтых зменаў у выпадку памылак, было вырашана паставіць наступныя патрабаванні:

- Павінна быць два падтыпа прывязак: перадумовы, якія выконваюцца перад запісам новага значэння, і постумовы, якія выконваюцца пасля запісу новага значэння.
- Калі адна з прывязак паведамляе пра памылку, другія прывязкі не выконваюцца і ўсе змены адмяняюцца.
- Прывязкі можна дадаваць і выдаляць у любы момант выканання сімуляцыі, акрамя тых адрэзкаў часу, калі выконваецца абнаўленне значэнняў, з якімі звязаны прывязкі. Пры гэтым павінны існаваць механізм, які дазволіў бы забараняць выдаленне «сістэмных» прывязак, якія ствараюцца базай дадзеных для ўнутраных патрэбаў, такіх як планіраванне абнаўлення індэксаў.
- Калі адбываецца змена некалькіх палёў картэжа адразу, напрыклад пры ўстаўцы картэжа ці выкананні UPDATE з больш чым адной SET аперацыяй, усе змены павінны быць запакаваны ў адзін пакет зменаў, тым самым дазволіўшы ці поўнае іх прымяненне, ці поўную адмену.

- Пры апрацоўцы пакета зменаў спачатку павінны выканацца ўсе перадумовы змяняемых палёў, потым павінны прымяніцца ўсе значэнні і толькі пасля гэтага павінны быць выкліканы постумовы.

Такім чынам аб’яднанне прывязак з пакетам з адной і большай колькасцю зменаў фарміруе транзакцыю і такую мадэль праверкі і прымянення зменаў можна назваць праграмнай рэалізацый транзакцыённай мадэлі памяці.

Трыгеры было вырашана рэалізаваць на аснове механізма прывязак: кожнае дзеяннне, да якога павінна быць магчымасць дадаць трыгер, павінна падтрымліваць даданне перадумоў і постумоў. Рэалізацыя механізму трыгераў базы дадзеных праз прывязкі дазволіць пазбегнуць дублікацыі вызначэнняў і коду, зрабіўшы працу з распрацоўваемай базай дадзеных больш трывіяльнай і лагічнай.

Я прааналізаваў сваю рэалізацыю транзакцыённай памяці і механізма прывязак, якія я рэалізаваў у межах вытворчай практыкі з тэмай «Асаблівасці праектавання архітэктуры кліента і сервера гульні для максімізацыі абароны ад узлома унутрагульнёвых транзакцый», і прыйшоў да вываду, што яны адпавядаюць пастаўленым патрабаванням і могуць выкарыстоўвацца ў распрацоўваемай эксперыментальнай базе дадзеных.

3.3. Базавыя тыпы і аперацыі над німі

Для таго, каб мець магчымасць рэалізоўваць аперацыі з палямі картэжаў, трэба спачатку вырашыць, якіх тыпаў могуць быць палі, а таксама выбраць механізм вызначэння тыпаў для гэтых палёў.

Самы трывіяльны і выкарыстоўваемы варыянт вызначэння тыпаў – вызначэнне на этапе выканання з дапамогай варыянтных тыпаў. Але вызначаць тыпы палёў можна і на этапе кампіляцыі, што павысіць хуткасць працы аперацый з палямі, але павялічыць час кампіляцыі і зробіць код больш складаным для чытання і разумення. Рэалізацыя вызначэння тыпаў на этапе кампіляцыі з’яўляецца нетрывіяльнай задачай, якая па аб’ёму хутчэй за ўсё заняла б увесь курсавы праект, таму было вырашана выкарыстаць механізм вызначэнне тыпаў на этапе выканання з дапамогай варыянтных тыпаў дадзеных.

Спачатку быў вызначаны набор тыпаў C++, якія могуць захоўвацца ў варыянтным тыпе:

- bool – булева значэнне.
- int64_t – 64-бітная цэлая лічба.

- uint64_t – 64-бітная неадмоўная цэлая лічба. Неабходныя абодва тыпа 64-бітных лічбаў з-за таго што вынікі некаторых арыфметычных аперацый для звычайных цэлых лічбаў і неадмоўных цэлых лічбаў адрозньваюцца.
- double – лічба з плаваючай кропкай.
- void* – сыры небяспечны ўказальнік на любое месца ў памяці.
- std::shared_ptr<void> – «разумны» указальнік з падлікам спасылак на любую структуру дадзеных.
- std::weak_ptr<void> – «разумны» слабы ўказальнік на любую структуру дадзеных.
- std::string – радок.

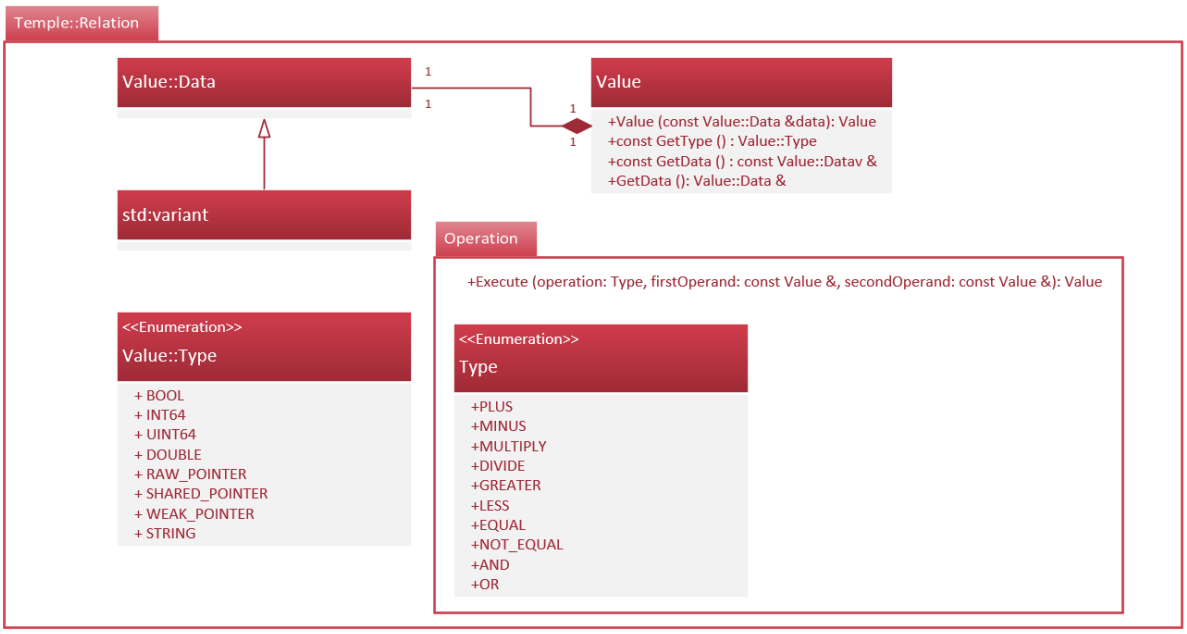
Для гэтага набору тыпаў было створаны варыянтны тып Value з дапамогай шаблона std::variant і ўтылітарнае пералічэнне Value::Type, якое дазваляе пры працы з варыянтным тыпам выкарыстоўваць не прамыя індэксы падтыпаў у варыянтным тыпе, а супадаючыя імянаваныя значэнні з утылітарнага пералічэння.

Для выкарыстання гэтых тыпаў у прэдыкатах фільтрацыі і функцыях падліку значэнняў неабходна задаць аперацыі паміж німі. Для пачатку было вырашана рэалізаваць толькі наступныя аперацыі з двума аргументамі:

- Складанне.
- Адніманне.
- Множанне.
- Дзяленне.
- Больш.
- Менш.
- Раўно.
- Не раўно.
- Лагічнае І.
- Лагічнае АБО.

У кодзе прылады гэты спіс аперацый захоўваецца як пералічэнне Operation::Type. Далей неабходна вызначыць тое, як гэтыя аперацыі будуць рэалізоўвацца. Вызначэнне тыпаў палёў адбываецца падчас выканання, таму і выбар рэалізацыі аперацыі таксама павінны адбывацца падчас выканання, за што адказвае функцыя Operation::Execute, якая атрымлівае на ўваход аперацыю Operation::Type і два аперанда Value. Выбар патрэбнай рэалізацыі Operation::Execute ажыцяўляе праз доступ да карты рэалізацый, якая з’яўляецца лінейным прадстаўленнем трохмернага масіва ўказальнікаў на функцыі-рэалізацыі канкрэтных аперацый паміж

канкрэтнымі падтыпамі варыянтнага тыпа. Калі аперацыя паміж двума дадзенымі тыпамі не вызначана, то выкідваецца выключэнне.



Выява 3.1 – UML-дыяграма рэалізацыі базавых тыпаў і аперацый.

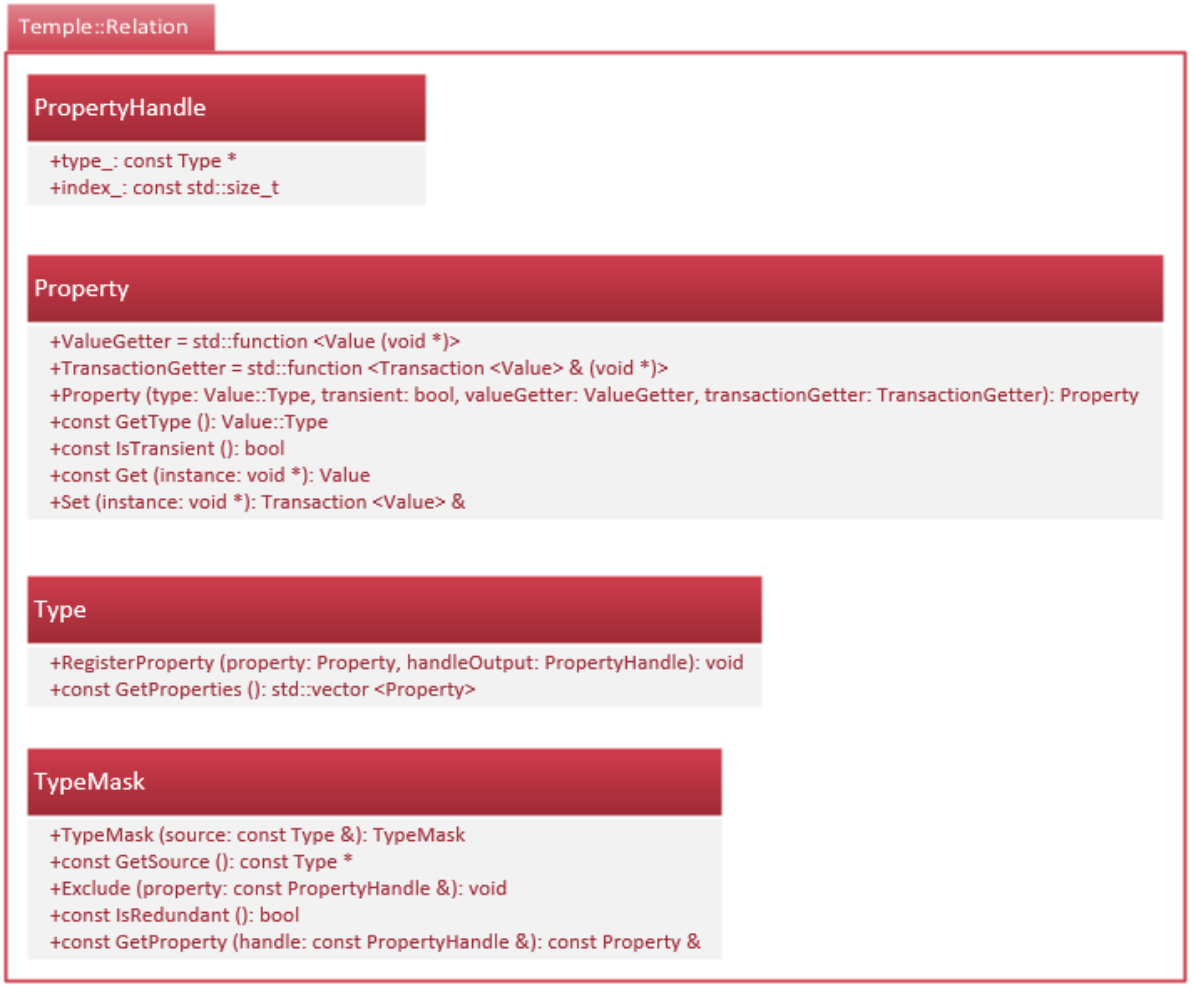
На першы погляд, запаўненне і падтрымка такой табліцы можа здавацца нерацыянальным рашэннем з-за памеру гэтай табліцы: на вызначаным наборы падтыпаў і аперацый у табліцы 640 элементаў, дзе кожны элемент указвае на функцыю, якую яшчэ трэба рэалізаваць. Рэалізацыя запаўнення такой табліцы «ў лоб» сапраўды займае многа радкоў коду і з'яўляецца зусім не элегантнай. Але з дапамогай прадуманнага выкарыстання макрасаў, шаблоннай функцыі і прынцыпа SFINAE (substitution failure is not an error, немагчымасць падстаноўкі не з'яўляецца памылкай) усю рэалізацыю запаўнення табліцы атрымалася умясціць у 150 радкоў коду. Выніковы код рэалізацыі выканання аперацый прадстаўлены ў дадатку А. З дапамогай гэтых жа прынцыпаў атрымалася умясціць рэалізацыю 200 юніт-тэстаў для Operation::Execute у 100 радкоў коду.

3.4. Сістэма метаінфармацыі для выкарыстоўваемых у базе дадзеных тыпаў

Як было вызначана раней, картэжы з'яўляюцца экземплярамі класаў ці структур C++, а вызначэнне тыпаў палёў і выбар рэалізацый аперацый над імі адбываецца падчас выканання праграмы. Таму неабходна для кожнага выкарыстоўваемага ў базе дадзеных тыпу захоўваць статычную метаінфармацыю, на аснове якой код базы дадзеных зможа чытаць і запісваць значэнні картэжаў па іх апісальніках. Падобны функцыянал мае

сістэма атрыбутаў у рухавіку Urho3D, таму было вырашана будаваць архітэктуру сістэмы метаінфармацыі для базы дадзеных падобным да сістэмы атрыбутаў Urho3D чынам.

Пачнем з дэманстрацыі дыяграмы класаў.



Выява 3.2 – UML-дыяграма рэалізацыі метаінфармацыі тыпаў.

Зыходны код рэалізацыі гэтых класаў прадстаўлены ў дадатку Б.

PropertyHandle – спецыяльная ўтылітарная структура дадзеных, якая ствараецца ў выніку рэгістрацыі поля аб’екта і з’яўляецца ўнікальным ідэнтыфікатарам гэтага поля. З дапамогай PropertyHandle можна атрымаць доступ да метаінфармацыі поля ў TypeMask ці звярнуцца да поля ў генерыруемай падчас выканання формуле.

Property – захоўвае ўсю метаінфармацыю для працы з канкрэтным полям структуры дадзеных. Захоўвае тып поля, флаг часовасці і функтары атрымання значэння поля і атрымання канструктара транзакцый, якія можна выкарыстоўваць для змены значэння поля. Падкрэслім два важных моманта:

- Поле структуры дадзеных можа мець любы тып, які можна абгарнуць у Value. Такім чынам яно застаецца звычайным полям і не стварае накладных расходаў, якія былі б, калі б усе палі, з якімі працуе база дадзеных, былі прымушаны мець тып Value.
- Праз перадачу канструктара транзакцый ужо на ўзроўні тыпаў вырашаецца праблема аб'яднання зменаў у пакеты і дадання прывязак да гэтых зменаў.

Type – захоўвае ўсю метаінфармацыю пра тып. На дадзены момант усёй патрэбнай ад тыпа метаінфармацыяй з’яўляюцца яго палі.

На дадзеным этапе распрацоўкі было вырашана не адкідваць класічныя аперацыі рэляцыйнай алгебры, калі іх рэалізацыя не становіцца прычынай залішняга ўскладнення архітэктуры базы дадзеных. Таму для падтрымкі аперацыі праекцыі спатрэбілася дадаць тып TypeMask, які падтрымлівае аперацыю «забывання» пра існаванне палёў у тыпа, тым самым маскіруючы частку палёў.

3.5. Фармат захавання інфармацыі ў адносінах

Вышэй былі вызначаны базавыя сродкі доступу да значэнняў у структурах дадзеных базы праз сістэму метаінфармацыі. Далей неабходна вызначыць унутраную структуру адносінаў у базе дадзеных.

Будзем лічыць, што экземпляр адносінаў рэляцыйнай алгебры складаецца з двух наступных частак:

- Галаўная частка – захоўвае такую інфармацыю пра адносіны, як набор метададзеных для працы з картэжамі. Гэта частка задаецца пры стварэнні адносінаў і не залежыць ад напаўнення адносінаў картэжамі.
- Цела адносінаў – захоўвае ўсе картэжы, што знаходзяцца ў адносінах.

Для падтрымкі ў базе дадзеных большасці стандартных рэляцыйных аперацый, лагічны картэж павінен мець магчымасць складацца з некалькіх розных структур дадзеных. Напрыклад, у сфарміраваных у выніку выканання JOIN адносінах могут быць палі з абаіх адносінаў, да якіх быў прыменены JOIN. Але падтрымка такіх кампазітных картэжаў значна ўскладніла б аптымізацыю базы дадзеных у найбольш выкарыстоўваемых выпадках, дзе картэж раўназначны структуры дадзеных. Таму ў распрацоўваемай базе дадзеных было вырашана адмовіцца ад падтрымкі такога тыпа картэжаў.

Такім чынам, галаўная частка адносінаў будзе рэалізавана як маска для тыпу, дадзеныя з якога захоўвае картэж – TypeMask. Далей гэты тып

будзе замяняцца псеўданімам Header. Картэжы было вырашана рэалізоўваць як абагульнены указальнік на любую структуру дадзеных – std::shared_ptr <void>. Далей гэты тып будзе замяняцца псеўданімам Entity. Было вырашана назваць гэты тып «існасцю» з-за таго, хоць ён і з'яўляецца рэалізацыяй лагічнага картэжа, ён не супадае па структуры з лагічным картэжам. Пры гэтым трэба заўважыць, што супадзенне назвы ніякім чынам не звязвае гэты тып з існасцямі ў шаблоне Існасць-Кампанент-Сістэма. Рашэнне выкарыстоўваць абагульнены ўказальнік замест унікальнага было прынята для таго, каб спрасціць рэалізацыю кэшыравання структур дадзеных, якое можа спатрэбіцца на практыцы. Рашэнне выкарыстоўваць безтыпавы ўказальнік void было прынята па наступных прычынах:

- Працягванне канкрэтных тыпаў праз шаблоны ў рэалізацыю адносінаў прывядзе да залішняга ўскладнення коду рэалізацыі без значных плюсаў у параўнанні з праверкай у публічных мадыфікуючых метадах.
- Праверку тыпаў зручней рэалізаваць у шаблонных публічных метадах, якія будуць правяраць уваходныя тыпы і перадаваць ужо бестыпавыя значэнні ва ўнутраную рэалізацыю, што дазволіць пазбегнуць пераўскладнення ўнутранай рэалізацыі.
- Стварэнне пустога базавага класа накшталт DatabaseObject не дало б яўнага павялічэння бяспечнасці рэалізацыі, а толькі ўскладніла б працу з базай дадзеных карыстальнікам, якім прыйшлося рабіць усе свае структуры нашчадкамі гэтага тыпа.

Усе картэжы было вырашана захоўваць у звычайным масіве std::vector, бо па патрабаваннях сферы выкарыстання неабходна трымаць усе дадзеныя ў памяці і забяспечваць хуткі доступ да іх па ўказальніку.

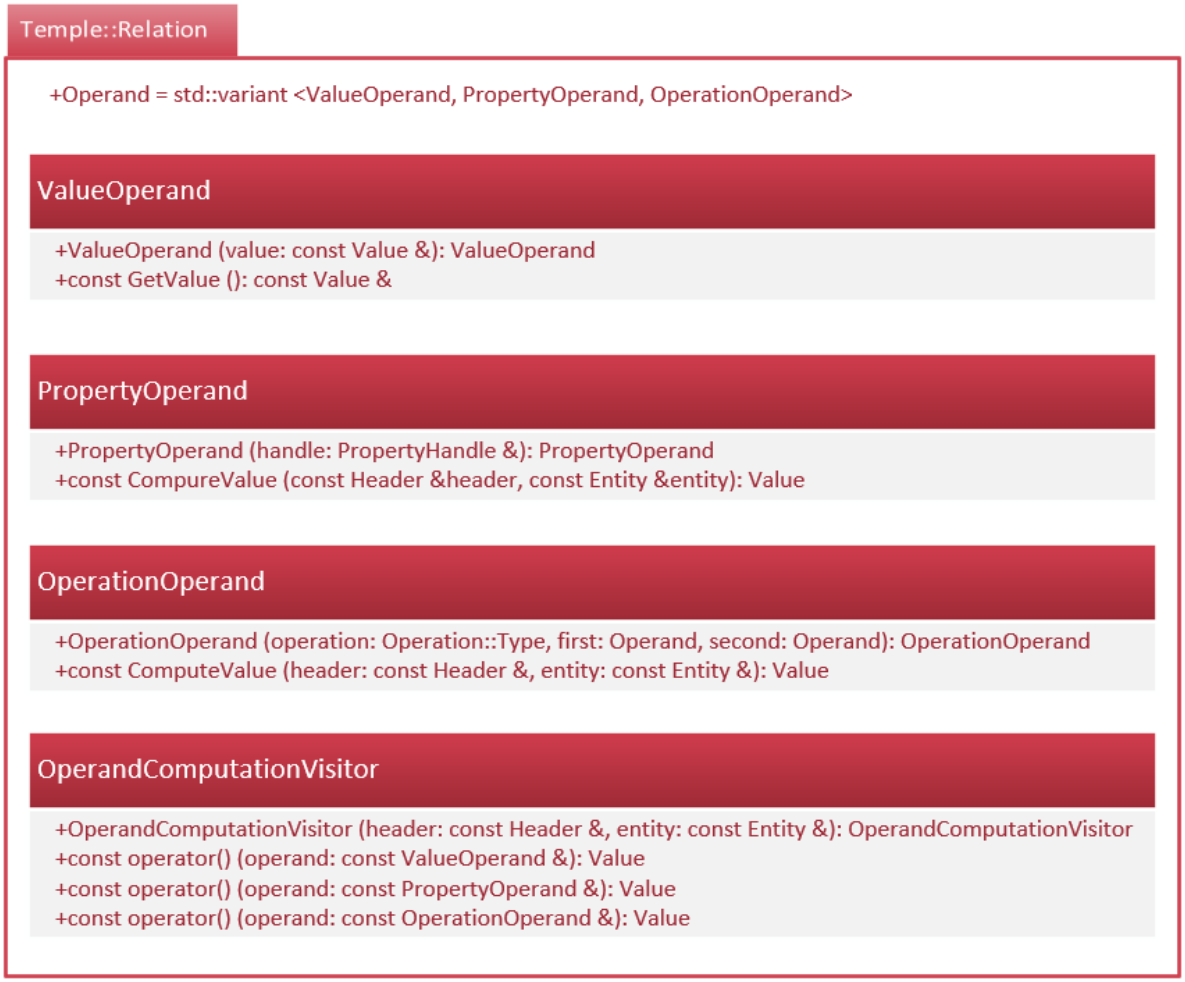
3.6. Будова формул для фільтрацыі і абнаўлення запісаў

Як было сказана раней, фарміраванне запытаў да базы дадзеных адбываецца падчас выканання. Гэта значыць, што трэба рэалізаваць структуры дадзеных, якія будуць фарміравацца падчас выканання і прадстаўляць механізм як для вылічэння значэнняў на аснове нейкага кантэкста, так і для аналізу формулы вылічэння з пошукам магчымых аптымізацый, напрыклад выбару індэкса на аснове WHERE-фільтра замест прахода па ўсіх картэжах.

Спачатку дадзім вызначэнне для кантэксту выканання вылічэнняў. Вылічэнні неабходны для WHERE-фільтраў і для падліку новых значэнняў у UPDATE-запытах. Для абодвух гэтых варыянтаў падыходзіць наступны

кантэкст: картэж, для якога выконваюцца вылічэнні, і галаўная частка адносінаў, да якіх прыналежыць гэты картэж.

Вызначыўшы кантэкст і мэты прымянення пяройдзем да апісання структуры рэалізацыі:



Выява 3.3 – UML-дыяграма рэалізацыі формул падліку значэнняў.

Зыходны код рэалізацыі гэтых класаў прадстаўлены ў дадатку В.

Было вырашана апісваць вылічэнні праз бінарнае дрэва, у якім усе вузлы – аперанды. Пры гэтым лісты – значэнні-канстанты Value ці аперацыі атрымання палёў з картэжу, а ўсе астатнія вузлы – аперацыі. Для рэалізацыі вылічэнняў спатрэбіліся наступныя тыпы аперандаў:

- ValueOperand – трывіяльны аперанд, які захоўвае ў сабе канстанту.
- PropertyOperand – аперанд, які захоўвае ў сабе PropertyHandle і можа знайсці дакладнае значэнне патрэбнага поля, атрымаўшы доступ да кантэкста: картэжа і галаўной часткі адносінаў.

- OperationOperand – аперанд, які захоўвае ў сабе тып аперацыі і два падпарадкаваных аперанда. Пры неабходнасці падліку значэння выконвае падлік значэнняў падпарадкаваных аперандаў з дапамогай кантэксту, пасля чаго прымяняе да вынікаў Operation::Execute.

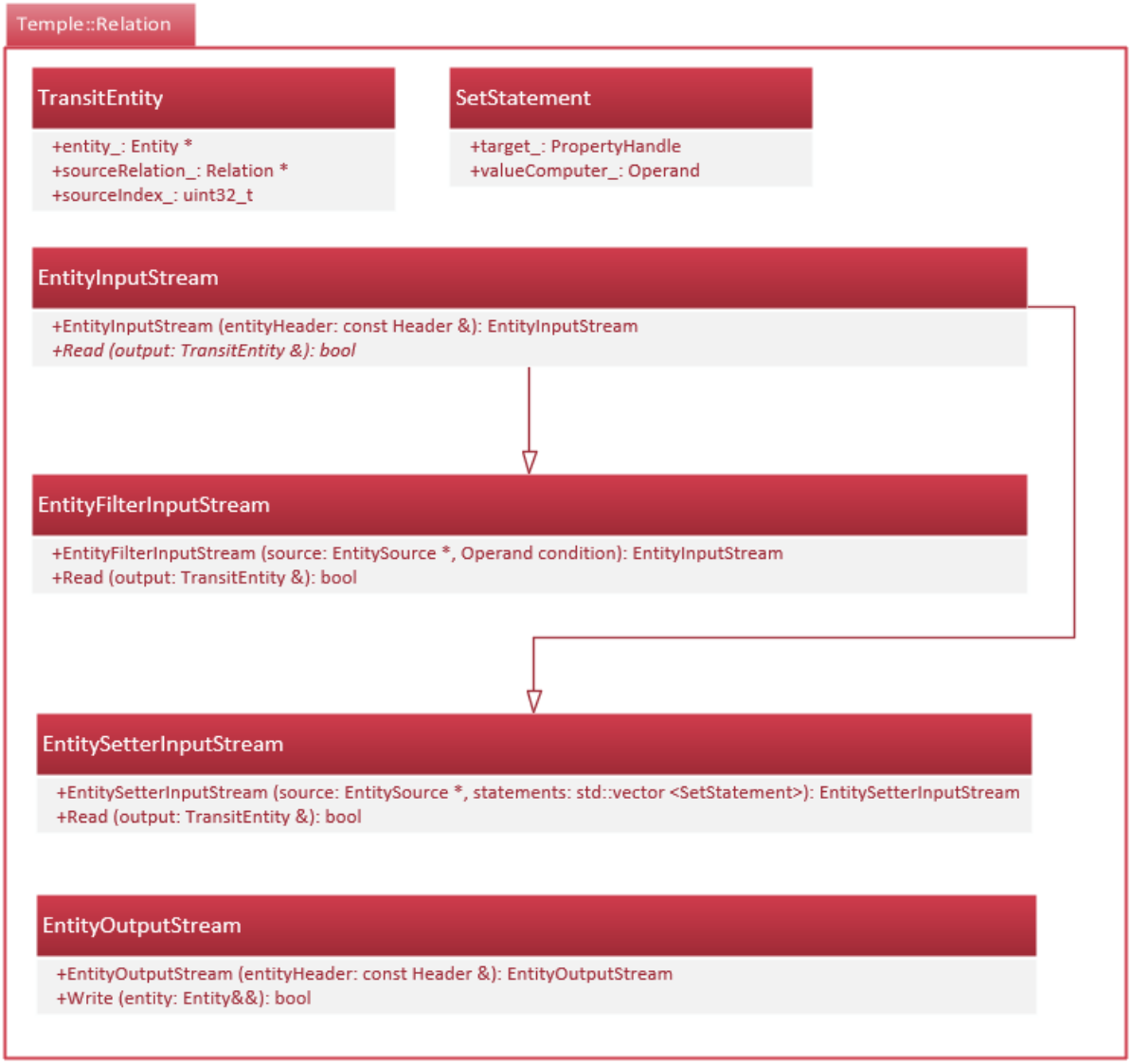
Заўважым, што дадаткова быў створаны варыянтны тып Operand, у якім можа захоўвацца любы з апісаных вышэй тыпаў аперандаў. Наяўнасць такога варыянтнага тыпа значна спрашчае рэалізацыю OperationOperand.

Для вылічэння значэнняў аперандаў была створана ўтылітарная структура OperandComputationVisitor, якая захоўвае кантэкст вылічэнняў і таксама рэалізуе шаблон наведвальніка для варыянтнага тыпа Operand. Рэалізацыя падліку значэнняў праз структуру-наведвальнік з кантэкстам вылічэння зрабіла трывіяльным атрыманне формул як аргументаў і выкарыстанне гэтых формул: метады, якім патрэбны формулы, могуць атрымліваць аргумент тыпа Operand і прымяняць наведвальнік OperandComputationVisitor, не правяраючы ўручную тып карнявога вузла формулы.

3.7. Рэалізацыя запытаў праз плыні

Плыні дадзеных з’яўляюцца моцным механізмам перадачы і апрацоўкі дадзеных, рэалізаваным у стандартных бібліятэках многіх моваў праграмавання, у тым ліку C++. Пры гэтым яны даволі падобныя на механізм курсораў з рэляцыйных баз дадзеных, якія таксама дазваляюць пакрокава атрымліваць дадзеныя і апрацоўваць іх пры атрыманні, не складваючы ў прамежкавы масіў. Такім чынам, дзякуючы ўніверсальнасці механізма плыняў і іх падабенству на курсоры рэляцыйных баз дадзеных, было вырашана будаваць рэалізацыю запытаў да адносінаў баз дадзеных на аснове механізма плыняў.

У гэтым падзеле будуць разгледжаны асноўныя тыпы плыняў, выкарыстоўваемыя для рэалізацыі запытаў, без прывязкі да рэалізацыі падтрымкі працы з плынямі ў адносінах, якая будзе разгледжана ў наступным падзеле. Пачнем агляд гэтых тыпаў з наступнай UML-дыяграмы:



Выява 3.4 – UML-дыяграмаў асноўных тыпаў рэалізацыі запытаў праз плыні.

Пачнем з агляду ўтылітарных тыпаў:

- TransitEntity – структура, выкарыстоўваемая для перадачы дадатковай інфармацыі разам з указальнікам на картэж пры чытанні картэжаў з адносінаў. Поле sourceRelation_ дазваляе заўсёды апазнаць, з якіх адносінаў быў атрыманы картэж, а поле sourceIndex_ з'яўляецца службовым значэннем, неабходным для выдалення картэжа з адносінаў, калі гэта будзе неабходна. Механізм выдалення будзе разгледжаны ў наступным падзеле.
- SetStatement – звязвае апісальнік поля структуры дадзеных картэжа з дрэвам падліку яго новага значэння. У кантэксце з галаўной часткай адносінаў і картэжам дазваляе падлічыць новае значэнне для поля.

EntityInputStream з’яўляецца базавым класам для плыняў, якія нейкім чынам, напрыклад чытаючы з адносінаў ці індэкса, атрымліваюць картэж, апрацоўваюць яго і перадаюць далей па ланцугу плыняў. Для канструіравання EntityInputStream у яго неабходна перадаць галаўную частку адносінаў, картэжы якіх будуць праходзіць праз плынь. Заўважым, што такія адносіны не абавязаны існаваць у рэальнасці, неабходна толькі каб структура прыходзячых картэжаў супадала з галаўной часткай. Гэта дазволіць у будучым адносна трывіяльна рэалізаваць JOIN – ён будзе аб’ядноўваць картэжы з другіх плыняў і перадаваць іх у аб’яднаным выглядзе пад новым аб’яднаным апісаннем галаўной часткі.

EntityFilterInputStream атрымлівае картэжы з любой іншай плыні і правярае вынік апрацоўкі перададзеным праз канструктар прэдыкатам, рэалізаваны ў выглядзе дрэва падліку значэння. Калі прэдыкат дае станоўчы вынік, то картэж перадаецца далей. Калі прэдыкат дае адмоўны вынік, то праверaны картэж прапускаецца і выконваецца запыт наступнага картэжу з падпарадкаванай плыні, паўтараючы гэта пакуль не будзе знойдзены адпаведны картэж ці пакуль падпарадкаваная плынь не паведаміць аб адсутнасці картэжаў.

EntitySetterInputStream атрымлівае картэжы з любой іншай плыні і прымяняе да ніх набор аперацый запісу новага значэння ў поле, якія перадаюцца праз канструктар як масіў SetStatement. Пры гэтым вынікі ўсіх аперацый падлічваюцца перад запісам, а запіс усіх новых значэнняў аб’ядноўваецца ў адзін транзакцыённы пакет зменаў.

Не цяжка заўважыць, што з дапамогай класаў EntityFilterInputStream і EntitySetterInputStream ужо можна рэалізоўваць такія SQL-запыты як «SELECT … WHERE ...» і «UPDATE … SET … WHERE …». Пры гэтым , дзякуючы ўніверсальнасці шаблона плыняў дадзеных, выкарыстанне гэтых класаў не абмяжоўваецца гэтымі тыпамі запытаў: іх можна лёгка прыстасаваць як часткі SELECT і UPDATE запытаў з выкарыстаннем JOIN.

Дадаткова падкрэслім наступныя моманты ў рэалізацыі чытаючых плыняў:

- Сярод рэалізацый EntityInputStream не было прадстаўлена чытаючых картэжы з адносінаў плыняў. Такія плыні будуць прадстаўлены ў наступным падзеле.
- EntityFilterInputStream і EntitySetterInputStream захоўваюць указальнікі на падпарадкаваныя плыні ў std::unique_ptr, тым самым дазваляючы каскадна зачыніць увесь ланцуг плыняў, калі канцавы вузел ланцуга перастаў быць патрэбным.

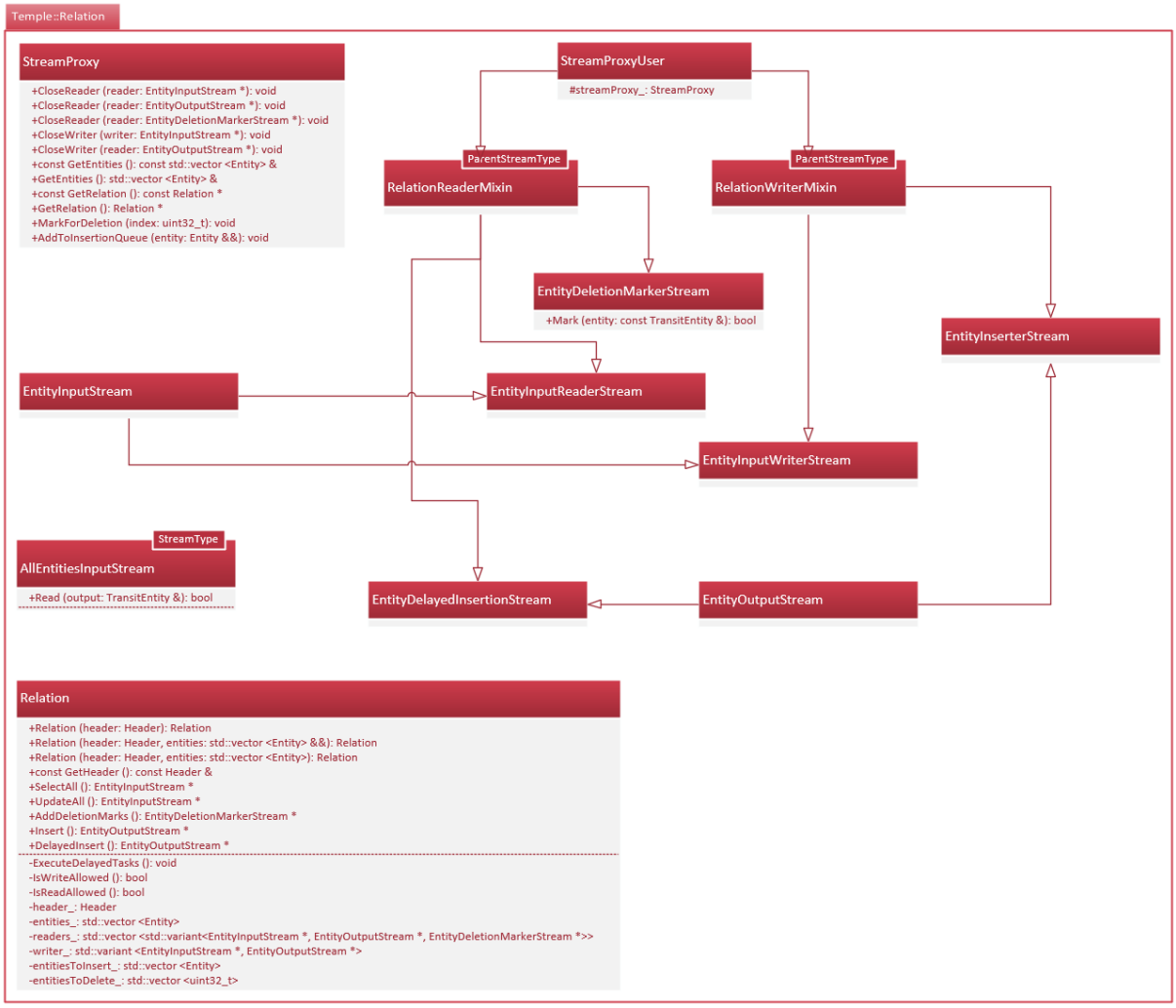
EntityOutputStream з’яўляецца базавым класам для плыняў, выконваючых зваротную ў параўнанні з EntityInputStream перадачу дадзеных: картэж адпраўляецца ў плынь як аргумент метада Write і далей перадаецца ці наступнаму EntityOutputStream, ці запісваецца ў канчатковае цэлявое сховішча. Заўважым, што метад Write вяртае булева значэнне, што дазваляе скончыць запіс тады, калі канчатковае сховішча больш не дазваляе запісваць у сябе. Таксама падкрэслім, што картэж перадаецца ў Write як rvalue, што дазваляе сінтаксічна паведаміць карыстальніку базы дадзеных аб тым, што значэнне будзе перанесена ў нейкае канчатковае сховішча. У гэтым раздзеле не будзем разглядаць канкрэтныя рэалізацыі такіх плыняў, бо на дадзены момант яны неабходны толькі для рэалізацыі INSERT-запытаў, што будзе абмеркавана ў наступным падзеле.

3.8. Інтэграцыя плынявай мадэлі ў адносіны.

На дадзеным этапе праектавання будзем лічыць, што ўсе аперацыі з адносінамі адбываюцца ў адной плыні выканання. Тым не менш, праектаваць узаемадзеянні з адносінамі было вырашана па класічнаму шаблону ўзаемадзення ў мнагаплынявых праграмах «чытачы-пісьменнікі» па наступных прычынах:

- На практыцы сітуацыі, падобныя станам гонкі ў мнагаплынявых прыладах, адбываюцца нават у аднаплынявых прыладах. Напрыклад, можа быць створана прывязка, якая выдаляе ўсіх персанажаў, у якіх значэнне ачкоў жыцця стала нулявым. Пры гэтым можа існаваць сістэма, якая праходзіцца па ўсіх персанажах і прымяняе да іх уздзеянні атрутаў, калі персанаж атручаны. Такім чынам выкананне прывязкі падчас выканання сістэмы атрутаў з неадкладным выдаленнем персанажа прывядзе да пашкоджання памяці і некаррэктнай працы гульні. Для адсячэння такіх сітуацый неабходна выкарыстоўваць спрошчаныя для захавання хуткасці выканання варыянты прымітываў сінхранізацыі.
- Для практычнага выкарыстання распрацоўваемай базы дадзеных з вялікай імавернасцю спатрэбіцца магчымасць доступа з розных плыняў. Трэба ўлічваць гэту магчымасць і ўжо на гэтым этапе праектаваць базу дадзеных так, каб такі пераход быў як мага больш простым.

Разгледзім дыяграму класаў, якія рэалізуюць плынявую мадэль для адносінаў базы дадзеных:



Выява 3.5 – UML дыяграма класаў, інтэгруючых адносіны з плынямі.

На дыяграме вышэй толькі класы Relation, EntityDeletionMarkerStream і вядомыя нам з мінулага падзела EntityInputStream з EntityOutputStream будуць бачны канчатковаму карыстальніку бібліятэкі. Астатнія класы з’яўляюцца прыватнай часткай рэалізацыі і не будуць даступны для знешняга выкарыстання.

Пачнем разбор дыяграмы з класа Relation, у першую чаргу звярнуўшы ўвагу на публічныя метады, якія ствараюць звязаныя з адносінамі плыні:

- SelectAll – стварае плыню з правамі чытача, якая дазволіць карыстальніку прайсціся па ўсім картэжам адносінаў, не змяняючы іх. Калі стварыць плыню немагчыма з-за наяўнасці актыўнага пісьменніка, то вяртаецца nullptr.
- UpdateAll – стварае плыню з правамі пісьменніка, якая дазволіць карыстальніку прайсціся па ўсім картэжам адносінаў, абнаўляючы іх значэнні. Калі стварыць плыню немагчыма з-за наяўнасці

іншага актыўнага пісьменніка ці актыўных чытачоў , то вяртаецца nullptr.

- AddDeletionMarks – стварае плыню з правамі чытача, у якую можна перадаць інфармацыю пра картэжы, якія неабходна выдаліць у той момант, калі з’явіцца магчымасць выдаляць картэжы. Такім чынам можна арганізаваць адкладзенае выдаленне праз перадачу ў створаную гэтым метадам плыню картэжаў, атрыманых праз SelectAll.
- Insert – стварае плыню з правамі пісьменніка, якая выконвае ўстаўку новых картэжаў у адносіны. Калі стварыць плыню немагчыма з-за наяўнасці іншага актыўнага пісьменніка ці актыўных чытачоў , то вяртаецца nullptr.
- DelayedInsert – стварае плыню з правамі чытача, якая перадае на захаванне адносінам картэжы без яўнага дадання картэжаў у адносіны. Спроба дадаць гэтыя картэжы будзе аўтаматычна выканана самой рэалізацыяй адносінаў тады, калі з’явіцца магчыманасць стварыць Insert-плыню. Калі стварыць плыню немагчыма з-за наяўнасці актыўнага пісьменніка, то вяртаецца nullptr.

На дадзены момант адсутнічаюць метады стварэнні паскораных індэксацыяй плыняў, бо было вырашана перанесці рэалізацыю індэксацыі на пераддыпломную практыку. Пры гэтым заўважым, што даданне індэксацыі амаль не зменіць плынявай структуры адносінаў: неабходна дадаць метады накшталт FilteredSelect і FilteredUpdate, якія будуць прымаць фільтры як дадатковыя параметры і на аснове аналізу гэтых фільтраў падбіраць найбольш адпаведны варыянт індэксацыі.

Разгледзім таксама некаторыя прыватныя метады і палі класа Relation:

- ExecuteDelayedTasks – выконвае адкладзеныя дзеянні, такія як выдаленне картэжаў, запыт на выдаленне якіх быў пакінуты праз створаныя з дапамогай метада AddDeletionMarks плыні, і ўстаўка новых картэжаў, запыт на ўстаўку якіх быў пакінуты праз створаныя з дапамогай метада DelayedInsert плыні.
- IsWriteAllowed і IsReadAllowed з’яўляюцца ўтылітарнымі метадамі, правяраючымі магчымасць стварэння плыняў-пісьменнікаў і плыняў чытачоў суадносна.
- У readers_ захоўваюцца указальнікі на ўсе актыўныя плыні-чытачы, калі такія існуюць.
- У writer_ захоўваецца ўказальнік на актыўную плыню-пісьменніка, калі такая існуе.

- У entitiesToInsert_ захоўваюцца картэжы, якія павінны быць дададзены ў ExecuteDelayedTasks.
- У entitiesToDelete_ захоўваюцца ўнутраныя ідэнтыфікатары пазначаных для выдалення картэжаў. Выдаленне будзе выканана у ExecuteDelayedTasks.

Пяройдзем да агляду класаў унутранай рэалізацыі ўзаемадзеяння з адносінамі праз плыні. Для таго, каб унутраныя рэалізацыі плыняў маглі ўзаемадзейнічаць з часткай прыватнага функцыянала адносінаў, быў створаны клас StreamProxy. Ён дазваляе выконваць наступныя дзеянні:

- Паведамляць праз зачыненне плыняў, працаваўшых з дадзенымі адносінаў.
- Атрымліваць прамы доступ да масіва картэжаў.
- Дадаваць значэнні ў entitiesToInsert_ і entitiesToDelete_.

Выкарыстоўваць проксі-аб’ект замест прямога доступа праз friend-механізм было вырашана з-за таго што friend механізм дае рэалізацыям плыняў залішне многа небяспечных магчымасцяў для змены адносінаў.

StreamProxyUser з’яўляецца ўтылітарным базавым класам для ўсіх класаў унутранай рэалізацыі, якім неабходна працаваць з адносінамі праз StreamProxy.

RelationReaderMixin і RelationWriterMixin з’яўляюцца ўтылітарнымі класамі, рэалізуючымі механізм RAII (resource acquisition is initialization, атрыманне рэсурса з’яўляецца ініцыялізацыяй) для ўнутраных рэалізацый плыняў, аўтаматычна выклікаючы StreamProxy::CloseReader і StreamProxy::CloseWriter суадносна. Для выбара карэктнага тыпу плыні, перадаваемай у метады StreamProxy, выкарыстоўваецца механізм CRTP (curiously recurring template pattern, «дзіўны» рэкурсіўны шаблон).

AllEntitiesInputStream з’яўляецца плыняй, дазваляючай прачытаць усе картэжы адносінаў. Такая магчымасць неабходна як для рэалізацыі метада Relation::SelectAll, так і для Relation::UpdateAll, якія ствараюць плыню-чытача і плыню-пісьменніка суадносна. Для таго, каб не ствараць з-за гэтага розныя рэалізацыі AllEntitiesInputStream, адну як чытача і адну як пісьменніка, было вырашана стварыць класы-дапаможнікі, EntityInputReaderStream і EntityInputWriterStream, аб’ядноўваючыя EntityInputStream з RelationReaderMixin і RelationWriterMixin суадносна, а ў AllEntitiesInputStream дадаць шаблонны параметр, ад якога выконваецца наследванне. Тады ў SelectAll ствараецца AllEntitiesInputStream<EntityInputReaderStream>, а ў UpdateAll – AllEntitiesInputStream<EntityInputWriterStream>.

Для маркіроўкі картэжаў на выдаленне была створана ўтылітарная плыня EntityDeletionMarkerStream, якая наследуецца ад RelationReaderMixin і прадстаўляе метад Mark, які прымае TransitEntity на выдаленне. Рашэнне прымаць TransitEntity замест прамога прыёма ўнутраных ідэнтыфікатараў, захоўваемых у TransitEntity было зроблена для таго, каб зрабіць выпадкі злоўжывання гэтай плынню больш відавочнымі, чым калі б у плынь перадаваліс проста лічбы.

Плынь EntityInserterStream, якая наследуе EntityOutputStream і RelationWriterMixin, займаецца ўстаўкай картэжаў і праз яе рэалізуецца метад Relation::Insert. А метад Relation::DelayedInsert ралізуецца з дапамогай плыні EntityDelayedInsertionStream, якая наследуе EntityOutputStream і EntityReaderMixin.

Такім чынам, з дапамогай апісаных вышэй класаў унутранай рэалізацыі атрымалася рэалізаваць выкананне запытаў Relation::SelectAll, Relation::UpdateAll, Relation::AddDeletionMark, Relation::Insert і Relation::DelayedInsert, пры гэтым улічваючы магчымасць пераходу на мнагаплынявы доступ і выкарыстоўваючы шаблон узаемадзеяння «чытачы-пісьменнікі».

3.9. Праектаванне і рэалізацыя індэксаў

Разгледзім праектаванне і рэалізацыю шчыльных ўпарадкаваных некластэрызуючых індэксаў у эксперыментальнай базе дадзеных.

Выява 3.6 – UML-дыяграма рэалізацыі ўпарадкаваных індэксаў.

Клас OrderedIndex захоўвае структуру дадзеных індэкса і прадстаўляе адносінам інтэрфэйс для ўзаемадзеення з індэксам. Разгледзім яго ўнутраныя палі:

- Поле relation_ захоўвае ўказальнік на адносіны, па якіх будуецца індэкс.
- Поле order_ з’яўляецца ўпарадкаваным спісам указальнікаў на запісы. У вызначэнні ўпарадкаванага некластэрызуючага індэкса было сказана, што ў індэксе таксама захоўваюцца значэнні індэксіруемых палёў, а не толькі ўказальнікі на запісы. Такі падыход дазваляе зэканоміць час пры выкарыстанні індэкса за кошт таго, што няма неабходнасці звяртацца да саміх запісаў, якія

могуць быць не загружаны ў памяць. Але з-за спецыфікі сферы выкарыстання эксперыментальная база дадзеных трымае ўсе дадзеныя ў аператыўнай памяці, таму няма сэнсу капіраваць іх у індэкс. Можа здавацца, што такі падыход прывядзе да праблем у тым выпадку, калі па індэксу ідзе курсор, якія абнаўляе значэнні індэксіруемых палёў, што можа зрабіць аперацыю абнаўлення індэкса пасля гэтых зменаў значна больш маруднай. Але, дзякуючы абранай раней сістэме кіравання доступам да адносінаў, па адносінах заўсёды можа ісці толькі адзін курсор з правам на змену палёў і, калі такі курсор існуе, то для гэтых адносінаў няма ніякіх іншых адчыненых курсораў. Тады, калі адбываецца змена індэксіруемых палёў, то індэкс можа напрамую запытаць пазіцыю ў спісе зменената запісу, а не шукаць яго пазіцыю па значэннях індэксіруемых палёў да прымянення зменаў.

- У полі pendingEntityUpdateIndices_ захоўваюцца пазіцыі запісаў у індэксе, індэксіруемыя палі якіх змяніліся. Рэалізацыя абнаўлення індэкса будзе разгледжана ніжэй.
- Поле baseProperties_ захоўвае спіс індэксіруемых палёў.
- У полі dependantStreams_ захоўваецца спіс усіх адчыненых плыняў, якія ўзаемадзейнічаюць з гэтым індэксам.

Разглядзеўшы ўнутраныя палі індэкса, звернем увагу на метады, патрэбныя адносінам для працы з індэксамі:

- Метад Open стварае плынь, якая дазваляе чытаць і мадыфікаваць запісы адносінаў у тым парадку, у якім яны знаходзяцца ў індэксе. Заўважым, што індэкс не адказвае за выдачу правоў на чытанне і абнаўленне палёў, гэты функцыянал застаецца ў адносінах і няма сэнсу дублікаваць яго ў індэксах. Зачыняецца плыня аўтаматычна пры выкліку яе дэсктруктара. Створаныя плыні дадаюцца ў dependantStreams_ і выдаляюцца пры зачыненні.
- Метад IsSafeToDrop дазваляе адносінам даведацца, ці можна бяспечна выдаліць гэты індэкс. Выдаленне з’яўляецца бяспечным толькі тады, калі няма ніводнай звязанай з індэксам плыні, што вызначаецца дзякуючы полю dependantStreams_.
- Метады OnEntityInserted і OnEntityDeleted дазваляюць індэксу рэагаваць на даданне і выдаленне запісаў, абнаўлячы order_. Пры гэтым патрабуецца, каб пры выкліку адносінамі гэтых метадаў не існавала ніводнай плыні па гэтаму індэксу. Гэта лёгка гарантаваць на ўзроўні адносінаў, таму што даданне і выдаленне запісаў у адносінах ужо працуе па падобнаму прынцыпу, а значыць лепей

дадаць перадумову выкліку метадаў, чым дубліраваць гэты механізм на ўзроўні індэкса.

- Пара метадаў RegisterEntityUpdate і CommitEntityUpdates створана для рэалізацыі механізма абнаўлення індэкса пры абнаўленні запісаў. RegisterEntityUpdate можа быць выкліканы адносінамі ў любы момант, калі адбылася змена ў полі запіса, якое выкарыстоўваецца індэксам. А CommitEntityUpdates гарантавана выклікаецца толькі тады, калі не існуе адчыненых плыняў, працуючых з гэтым індэксам. Такім чынам RegisterEntityUpdate паведамляе аб змене запіса, пакідаючы індэксу выбар: прымяняць змену зараз ці адкласці абнаўленне парадку на потым. Калі на дадзены момант як мінімум адна плынь выкарыстоўвае індэкс, то абнаўленне парадку зрабіла б гэту плынь некаррэктнай, таму ў такім выпадку змены павінны быць адкладзены. CommitEntityUpdates як раз такі дазваляе прымяніць адкладзеныя змены, гарантуючы што ў дадзены момант гэта можна зрабіць бяспечна. Такім чынам, у RegisterEntityUpdate рэгіструюцца і, пры немагчымасці выканання абнаўлення ў гэты момант, запісваюцца ў поле pendingEntityUpdateIndices_ пазіцыі ў спісу змененых запісаў, а ў CommitEntityUpdates счытваюцца і абнаўляюцца ўсе запісаныя ў pendingEntityUpdateIndices_ пазіцыі.

Апісаныя вышэй метады могуць быць выкліканы толькі адносінамі, у якіх знаходзіцца экземпляр індэкса. Знешнія кліенты могуць звяртацца да індэкса толькі з дапамогай спецыяльных метадаў, якія былі дададзены ў адносіны:

- CreateOrderedIndex стварае шчыльны ўпарадкаваны некластэрызуючы індэкс па дадзенаму набору палёў, калі індэкс па такому набору палёў яшчэ не існуе. Метад вяртае ўнікальны ў межах адносінаў ідэнтыфікатар створанага ці існуючага індэкса. На дадзены момант для спрашчэння рэалізацыі было вырашана, што ўсе індэксы будуць упарадкоўваць запісы ў парадку ўзрастання.
- DropIndex спрабуе выдаліць індэкс па яго ўнікальнаму ідэнтыфікатару і ў якасці адказа вяртае сцяг паспяховасці выдалення. Калі індэкс нельга выдаліць бяспечна, то выдаленне не адбываецца і лічыцца, што аперацыя не была завершана паспяхова.
- SelectAllFromIndex і UpdateAllFromIndex аналагічны ўжо існуючым метадам SelectAll і UpdateAll, але вяртаюць плыні, якія перадаюць запісы ў парадку, які задаецца індэксам з

перададзеным у якасці аргумента ўнікальным ідэнтыфікатарам. Калі індэкса з такім ідэнтыфікатарам не існуе, вяртаецца nullptr.

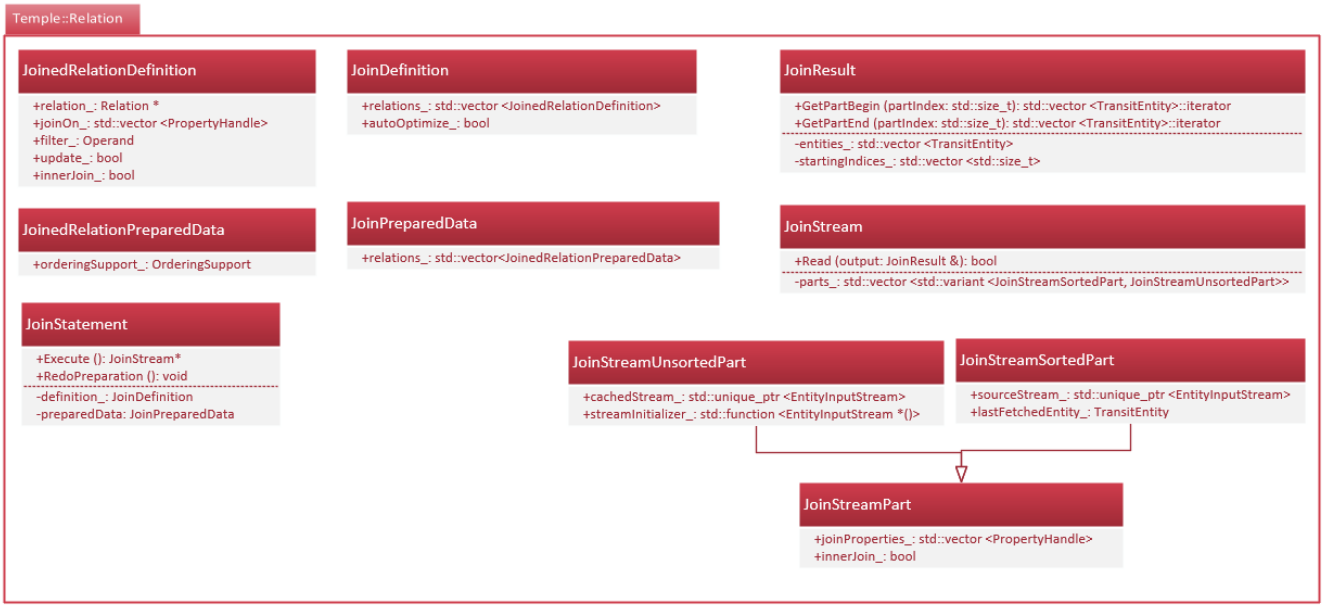
- SelectAllOrdered і UpdateAllOrdered прымаюць на ўваход спіс палёў, па якіх павінны быць адсартаваны запісы, і, аналагічна SelectAllFromIndex і UpdateAllFromIndex, вяртаюць плыні, запісы ў якіх адсартаваны па перададзеных палях. Вяртаемая плынь можа быць плынню індэкса, калі ў адносінах ёсць адпаведны індэкс. Калі ёсць індэкс па ненулявому прэфіксу спіса палёў, то ствараецца плынь, якая выкарыстоўвае плынь з індэкса і дасартыроўвае атрыманыя з яе запісы. Калі паскарэнне праз індэксы немагчыма, то адбываецца поўная сартыроўка. Заўважым, што калі атрымліваць адсартыраваную плынь запісаў неабходна вельмі часта, напрыклад кожны крок сімуляцыі, то не раіцца выкарыстоўваць гэтыя метады, бо нават пры наяўнасці поўнасцю адпаведнага індэкса пры выкліку затрачваецца час на пошук гэтага індэкса.
- QueryOrderingSupport, аналагічна SelectAllOrdered і UpdateAllOrdered, выконвае пошук найбольш аптымальнага падыходу да стварэння плыні запісаў, якая вяртае запісы у парадку сартыроўкі па перададзеных палях, але замест таго каб пасля гэтага стварыць такую плынь, вяртае вынік гэтага пошука ў структуры OrderingSupport, якая складаецца з трох палёў: ідэнтыфікатар bestIndex_, масівы палёў sortedProperties_ і unsortedProperties_. Калі запыт нельга паскорыць праз індэкс, то ў bestIndex_ будзе запісаны спецыяльны некарэктны ідэнтыфікатар індэкса, sortedProperties_ застанецца пустым, а unsortedProperties_ будзе захоўваць усе запытаныя палі. Калі запыт можна хаця б часткова паскорыць праз нейкі індэкс, то ў bestIndex_ будзе запісаны ідэнтыфікатар гэтага індэкса, sortedProperties_ будзе захоўваць спіс палёў, сартыроўку па якіх гарантуе індэкс, а unsortedProperties_ – спіс палёў, дасартыроўваць па якіх павінна дадатковая плынь. Калі запыт можна паскорыць праз розныя індэксы, то выбіраецца лепшы. Такім чынам QueryOrderingSupport дазваляе падлічыць і захаваць інфармацыю, патрэбную для хуткага адкрыцця такой жа плыні, як плыні, якія адчыняцца праз SelectAllOrdered і UpdateAllOrdered, а гэта неабходна для тых запытаў, якія выконваюцца ў кожным кроку сімуляцыі.

Пры рэалізацыі індэксаў спатрэбілася стварыць дадатковыя тыпы плыняў, якія перадаюць запісы:

- Плынь IndexInputStream чытае запісы са стварыўшага яе індэкса ў тым парадку, у якім яны знаходзяцца ў гэтым індэксу. Звычайна плыні такога тыпу ствараюцца індэксамі і вяртаюцца як вынік выкліку метада Open.
- Шаблонная плынь WrapperEntityInputStream з'яўляецца утылітарнай плынню для дадання правоў на чытанне ці рэдагаванне запісаў у адносінах да падпарадкаванай плыні, якая выкарыстоўвае падобны да шаблоннай плыні AllEntitiesInputStream механізм, каб прыняць ролю плыні з правам чытання ці плыні з правам рэдагавання. Гэта плынь дазваляе звязваць трывіяльныя ўнутраныя плыні, накшталт плыняў з індэксаў, з рэалізаваным у адносінах механізмам правоў.
- Плынь SorterInputStream рэалізоўвае механізм дадання сартыроўкі па патрэбных палях да нейкай падпарадкаванай плыні, запісы ў якой ужо могуць быць адсартыраваны па нейкіх палях. У SorterInputStream захоўваецца масіў палёў alreadySortedProperties_, у якім указана, па якіх палях ужо адсартыраваны запісы ў падпарадкаванай плыні, і масіў палёў additionalSortingProperties_, сартыроўку па якіх трэба прымяняць для запісаў, роўных па палях з alreadySortedProperties_. У адрозненні ад больш простых плыняў, якія аперыруюць з асобнымі запісамі, плынь SorterInputStream аперыруе з блокамі запісаў, якія роўныя адзін аднаму па палях з alreadySortedProperties_, і сартыруе кожны блок па палях additionalSortingProperties_, таму ёй патрэбны дадатковы буфер для апрацоўваемых запісаў inputBuffer_. Заўважым, што alreadySortedProperties_ можа быць пустым, у такім выпадку плынь выканае сартыроўку ўсіх запісаў з падпарадкаванай плыні па additionalSortingProperties_.

3.10. Праектаванне і рэалізацыя аперацыі злучэння

Разглядзеўшы рэалізацыю індэксацыі, можна перайсці да апісання рэалізацыі аперацыі злучэння, якая ў залежнасці ад абставінаў будзе выконвацца праз алгарытм зліцця ці праз злучэнне з ўкладзеным цыклам.



Выява 3.7 – UML-дыяграма рэалізацыі аперацыі злучэння.

Структура JoinDefinition апісвае запыт на злучэнне запісаў з адносінаў і мае наступныя палі:

- Поле relations_ з’яўляецца масівам са сцэнарамі атрымання запісаў з кожнага выкарыстоўваемага экземпляра адносінаў.
- Калі сцяг поля autoOptimize_ выстаўлены, то пры падрыхтоўцы запыту могуць ствараца дадатковыя індэксы для аптымізацыі выканання запыту. У дадзенай рэалізацыі для кожнага экземпляра адносінаў ствараецца ўпарадкаваны некластэрызуючы індэкс па ўсіх палях, якая выкарыстоўваюцца для злучэння па аперацыі ¿, у тым выпадку, калі атрыманне запісаў з гэтага экземпляра нельга паскорыць індэксам, які пакрывае як мінімум палову выкарыстоўваемых палёў.

Разгледзім палі структуры JoinedRelationDefinition, у якой апісваецца сцэнар працы з канкрэтным экземплярам адносінаў у межах запытанай аперацыі злучэння:

- Поле relation_ захоўвае спасылку на адносіны, з якіх будуць запытвацца запісы.
- У полі joinOn_ захоўваецца спіс палёў, якія будуць выкарыстоўвацца для злучэння па аперацыі ¿. Пры злучэнні з запіса бярэцца картэж значэнняў з гэтых палёў, які далей параўноўваецца з аналагічнымі картэжамі, якія прадстаўляюць запісы з другіх адносінаў. Відавочна, што ва ўсіх JoinedRelationDefinition у межах JoinDefinition колькасць палёў у joinOn_ і іх тыпы павінны супадаць, інакш гэта будзе лічыцца

памылкай і падчас выканання сімуляцыі такі запыт не будзе выкананы.

- Поле filter_, па аналогіі з плынню EntityFilterInputStream, захоўвае дрэва-прэдыкат, якое дазваляе прапускаць непатрэбныя запісы з адносінаў па дадатковаму фільтру.
- Калі сцяг update_ усталяваны, то запісы з гэтых адносінаў будуць запытвацца з правам на рэдагаванне. Інакш запісы будуць запісвацца з правам на чытанне.
- Флаг innerJoin_ кажа, што вынік злучэння не з'яўляецца карэктным і не павінен быць перададзеным карыстальніку ў тым выпадку, калі ў яго не ўваходзіць хаця б адзін запіс з гэтых адносінаў. Такі падыход дазваляе эмуліраваць паводзіны звычайнага правайдэра кампанентаў у шаблоне Існасць-Кампанент-Сістэма, які не перадае існасць у тым выпадку, калі ў ёй адсутнічае адпаведны кампанент. Сцяг было вырашана назваць так, бо калі ва ўсіх сцэнарах ён усталяваны, то аперацыя злучэння стане ўнутранай.

Але для таго, каб эфектыўна выконваць злучэнні кожны крок сімуляцыі, неабходна падрыхтаваць запыты для гэтага. У структуры JoinedRelationPreparedData захоўваецца вынік падрыхтоўкі запыта для адных адносінаў з JoinedRelationDefinition, атрыманы праз выклік метада QueryOrderingSupport у адносінаў. А структура JoinPreparedData захоўвае масіў JoinedRelationPreparedData для ўсіх сцэнароў, які існавалі ў JoinDefinition.

Клас JoinStatement аб'ядноўвае ў сабе структуры JoinDefinition і JoinPreparedData. У канструктары гэты клас атрымлівае інфармацыю пра запыт у фармаце структуры JoinDefinition, якую капіруе сабе, пасля чаго падрыхтоўвае яе да выканання, запаўняючы унутраны экземпляр JoinPreparedData і ствараючы дадатковыя індэксы, калі гэта неабходна і калі ўсталяваны сцяг autoOptimize_ у JoinDefinition. Метад RedoPreparation дазваляе ў любы момант скінуць JoinPreparedData і паўтарыць падрыхтоўку. А метад Execute спрабуе стварыць плынь JoinStream для выканання аперацыі злучэння і вяртае яе ў выпадку поспеху ці nullptr пры немагчымасці яе стварэння.

Важным адрозненнем JoinStream ад звычайных плыняў з'яўляецца тое, што вяртае яна не TransitEntity, а спецыяльную структуру JoinResult, у якой захоўваюцца ўсе злучаныя па аднаму картэжу значэнняў палёў запісы. Разглядзім падрабязней яе палі:

- У палі entities_ захоўваюцца ўсе злучаныя запісы. Было вырашана захоўваць спасылкі на ўсе злучаныя запісы ў адным масіве, а не

стварацъ па масіву для запісаў з кожных адносінаў, для таго каб пазбегнуць фрагментацыі памяці і лішніх алакацый, бо ў большасці выпадкаў у адным JoinResult будзе знаходзіцца па аднаму запісу для кожных адносінаў.
- Поле startingIndices_ захоўвае масіў, у якім з кожнымі адносінамі з JoinDefinition супастаўляецца індэкс масіва entities_, які ўказвае на першы запіс з гэтых адносінаў.

З улікам таго, што працуючыя з JoinResult знешнія алгарытмы павінны мець доступ да JoinDefinition запыта, з якога была створана плынь, JoinResult мае наступныя публічныя метады:

- GetPartBegin прыймае індэкс сцэнара атрымання запісаў JoinedRelationDefinition з JoinDefinition і вяртае ітэратар, які ўказвае на пачатак адрэзка ў entities_, у якім захоўваюцца атрыманыя з гэтага сцэнара запісы.
- GetPartEnd прымае аналагічны індэкс і вяртае ітэратар, які ўказвае на канец аналагічнага адрэзку. Калі сцэнар не вярнуў запісаў, то вынікі GetPartBegin і GetPartEnd супадаюць.

У самім JoinStream захоўваецца толькі адно поле-масіў parts_, у якім захоўваюцца станы працы з кожнымі адносінамі, апісанымі ў JoinDefinition. У залежнасці ад таго, ці могуць адносіны эфектыўна прадставіць плыню запісаў, ўпарадкаваную па выкарыстоўваемых для злучэння запісах, стан для гэтай плыні будзе мець тып JoinStreamSortedPart ці JoinStreamUnsortedPart. Абодва тыпы наследуюць агульны тып JoinStreamPart з масівам joinProperties_, аналагічным полю joinOn_ у JoinedRelationDefinition, і сцягам innerJoin_, які бярэцца таксама з JoinedRelationDefinition.

JoinStreamSortedPart апісвае стан працы з адносінамі, якія змаглі стварыць эфектыўную плынь, якая перадае запісы ў патрэбным парадку. У ёй захоўваецца ўказальнік на гэту плыню sourceStream_ і апошні перададзены ёй запіс lastFetchedEntity_, выкарыстанне якога будзе растлумачана далей пры апісанні алгарытма. Калі плыня стала пустой, то ўказальнік sourceStream_ зануляецца і значэнне lastFetchedEntity_ не правяраюцца.

JoinStreamUnsortedPart наадварот апісвае стан працы з адносінамі, для якіх не атрымалася стварыць эфектыўную плынь. У ім захоўваецца лямбда streamInitializer_, якая па запыце стварае плыню, якая вяртае запісы адносінаў у любым парадку, і закэшыраваны ўказальнік на адну такую плынь cachedStream_. Кэшыраванне адной такой плыні неабходна для таго,

каб другія лагічныя часткі праграмы не маглі перахапіць доступ да адносінаў і тым самым заблакіраваць выкананне плыні злучэння.

Пры стварэнні плыні JoinStream на базе дадзеных з JoinStatement запаўняецца масіў parts_, выконваецца запыт першых запісаў з JoinStreamSortedPart і ствараюцца першыя закэшыраваныя плыні ў JoinStreamUnsortedPart. Палі filter_ і update_ у JoinedRelationDefinition улічваюцца на гэтым этапе і ўплываюць на тое, як будуць стварацца плыні для частак. Далей разгледзім у спрошчаным апісанні алгарытм выканання аперацыі Read у JoinStream:

- Пакуль як мінімум адна з частак у parts_ знаходзіцца ў працаздольным стане, выконваецца цыкл:
  - Калі ёсць хаця б два JoinStreamSortedPart у працаздольным стане:
    - Сярод lastFetchedEntity_ у JoinStreamSortedPart шукаем частку з найменшым картэжам па joinProperties_.
    - Калі ёсць некалькі JoinStreamSortedPart з найменшым картэжам, то злучаем іх запісы і шукаем па алгарытму злучэння з укладзеным цыклам запісы з працаздольных JoinStreamUnsortedPart, з якімі іх можна злучыць. З выкарыстаных JoinStreamSortedPart счытваюцца наступныя запісы, а атрыманнае злучэнне запісаў вяртаецца як вынік аперацыі Read.
    - Калі найменшы картэж ёсць толькі ў адным JoinStreamSortedPart, счытваем з яго наступны запіс і працягваем цыкл.
  - Інакш:
    - Калі ёсць адзін JoinStreamSortedPart у працаздольным стане, то бяром яго запіс lastFetchedEntity_ як «базавы» для злучэння і чытаем наступны. Інакш першы JoinStreamUnsortedPart у працаздольным стане назавем базавай часткай і счытаем з яго «базавы» запіс. Далей ва ўсіх астатніх працаздольных JoinStreamUnsortedPart шукаюцца запісы, якія можна злучыць з «базавым», і вяртаецца вынік злучэння.
- Усе злучэнні выкананы, плынь не можа паспяхова вярнуць новае злучэнне.

Пры гэтым усе часткі пры стварэнні лічацца працаздольнымі і губляюць гэты статус пры наступных умовах:

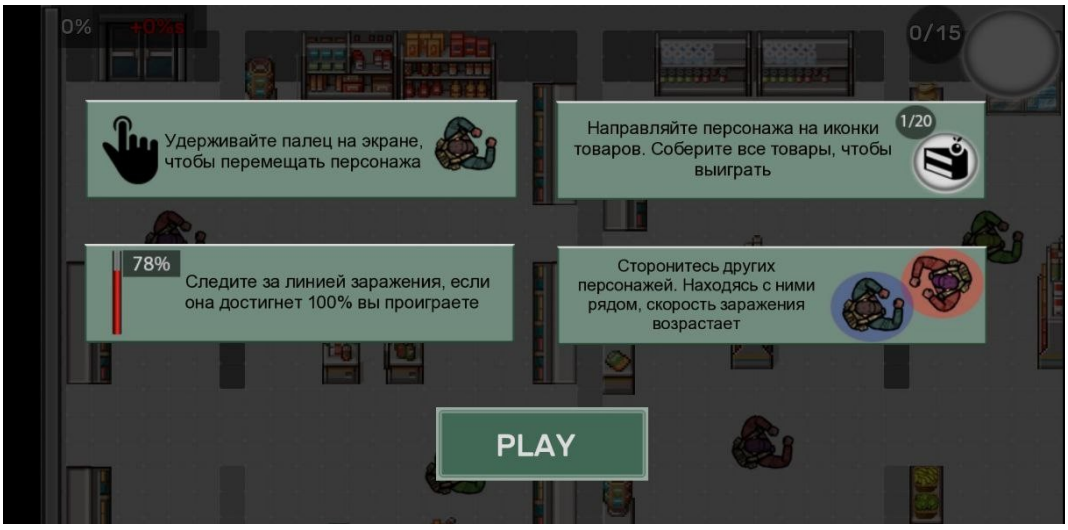
- Частка JoinStreamSortedPart становіцца непрацаздольнай тады, калі з яе плыні больш нельга счытваць запісы. Такія непрацаздольныя часткі маркіруюцца зануленнем sourceStream_.
- Частка JoinStreamUnsortedPart становіцца непрацаздольнай тады, калі пры счытванні з яе плыні «базавага» запісу гэты запіс апынуўся апошнім у плыні. Такія непрацаздольныя часткі маркіруюцца зануленнем streamInitializer_.

Таксама заўважым, што ва ўсіх выпадках чытання з JoinStreamUnsortedPart, акрамя чытання «базавага» запісу, пасля счытвання патрэбных запісаў з гэтай часткі выкарыстаная плынь выдаляецца і замяняецца новай.

Падкрэслім, што ў апісаным вышэй спрошчаным алгарытмы не ўлічваўся сцяг innerJoin_, якія дазваляе аптымізаваць некаторыя выпадкі, але ўскладняе алгарытм. Гэты сцяг выкарыстоўваецца ў рэалізацыі злучэння, але быў прапушчаны ў апісанні алгарытма для спрашчэння гэтага апісання.

3.11. Агляд распрацаванай базы дадзеных

Для праверкі магчымасці практычнага выкарыстання распрацаванай эксперыментальнай базы дадзеных у якасці пастаўшчыка існасцяў сістэмам было вырашана распрацаваць аднолькавую трывіяльную гульню ў двух варыянтах: з дапамогай распрацаванай базы дадзеных і з дапамогай EnTT – вядомай рэалізацыі шаблона Існасць-Кампанент-Сістэма з адчыненых зыходным кодам. У распрацаванай для праверкі гульні задачай гульца з’яўляецца пошук прадуктаў у супермаркеце падчас эпідэміі: каб не заразіцца і паспяхова завяршыць гульню, неабходна мінімізаваць кантакты з кіруемымі штучным інтэлектам гульцамі.



Выява 3.8 – Экран з тлумачэннем правілаў праверачнай гульні.

Выява 3.9 – Першы здымак экрану з праверачнай гульні.

Выява 3.10 – Другі здымак экрану з праверачнай гульні.

Для адмалёўкі гульнёвага сусвету, інтэрфэйсу карыстальніка і працы з рэсурсамі выкарыстоўваўся рухавік з адчыненым зыходным кодам Urho3D. Для апрацоўкі двумернай фізікі выкарыстоўваўся фізічны рухавік з адчыненым зыходным кодам Box2D. З дапамогай шаблона Існасць-Кампанент-Сістэма была рэалізавана ўся гульнёвая логіка, у тым ліку:

- Кіраванне пакупнікамі з штучным інтэлектам.
- Апрацоўка інфармацыі пра фізічныя сутыкненні.
- Апрацоўка гульнёвых падзей і індыкатараў, напрыклад індыкатара заражэння і падбору прадметаў.
- Апрацоўка дзеянняў карыстальніка.

Пералічым найбольш значныя са знойдзеных пры апісанай праверцы плюсаў і мінусаў распрацаванай эксперыментальнай базы дадзеных:

- Больш абагульнены механізм звязвання ў эксперыментальнай базе дадзеных дазваляе выконваць звязванне па любых дадзеных, а не толькі па існасцях. Такім чынам з'яўляецца магчымасць афармляць некаторыя дадзеныя як асобныя адзінкі, а не як кампаненты. Такі падыход не ўваходзіць у класічнае разуменне шаблона Існасць-Кампанент-Сістэма, але ён дазваляе зрабіць апрацоўку некаторых гульнёвых падзей больш інтуітыўнай. Напрыклад, знойдзеныя пры апрацоўцы фізічным рухавіком сутыкненні можна захоўваць як асобныя інфармацыйныя адзінкі, і гэта не прывядзе да праблем пры апрацоўцы іх сістэмамі. А класічны варыянт захавання гэтых сутыкненняў як кампанентаў закранутых існасцяў прывеў бы да дублікацыі дадзеных і павялічэння складанасці логікі.
- Праграмны інтэрфэйс узаемадзеяння з распрацаванай базай дадзеных з'яўляецца адчувальна менш зручным і лаканічным, чым інтэрфэйс узаемадзеяння з EnTT. Гэта чаканы мінус, бо распрацаваная база дадзеных з'яўляецца эксперыментальным праектам, які яшчэ не падрыхтоўваўся да камерцыйнага выкарыстання, у адрозненні ад EnTT.
- Эксперыментальная база дадзеных мае ўбудаваны механізм транзакцый, які дазваляе сумяшчаць шаблон Існасць-Кампанент-Сістэма з некаторымі падыходамі класічнага рэактыўнага праграмавання ў тых выпадках, калі гэта сапраўды неабходна ці дазваляе спрасціць гульнёвую логіку.
- Распрацаваная база дадзеных выконвае задачу пастаўкі існасцяў сістэмам крыху больш марудна, чым EnTT. Гэта можна растлумачыць тым, што праект з'яўляецца эксперыментальным і яго рэалізацыя яшчэ не аптымізавалася. Напрыклад, адным з перспектыўных шляхоў для аптымізацыі з'яўляецца пазбаўленне ад выкарыстання палімарфізма часу выканання ў логіцы плыняў, бо адной з прычын хуткасці EnTT як раз з’яўляецца выкарыстанне толькі палімарфізму часу кампіляцыі.
- У адрозненні ад EnTT, распрацаваная база дадзеных гарантуе стабільнасць указальнікаў на кампаненты, што дазваляе спрасціць інтэграцыю іншых модуляў, якія не праектаваліся пад працу з шаблонам Існасць-Кампанент-Сістэма.
- Дапаўненне рэалізаванай эксперыментальнай базы дадзеных іншымі механізмамі рэляцыйных баз дадзеных дазволіць выкарыстоўваць яе і ў другіх, больш экзатычных, варыянтах

шаблона Існасць-Кампанент-Сістэма, напрыклад у так званай падзейна-арыентаванай сімуляцыі. Многія вядомыя рэалізацыі шаблона Існасць-Кампанент-Сістэма, такія як EnTT, алгарытмічна завязаны на класічны варыянт шаблона Існасць-Кампанент-Сістэма, што робіць неэфектыўным іх выкарыстанне ў эксперыментальных варыянтах шаблона Існасць-Кампанент-Сістэма.

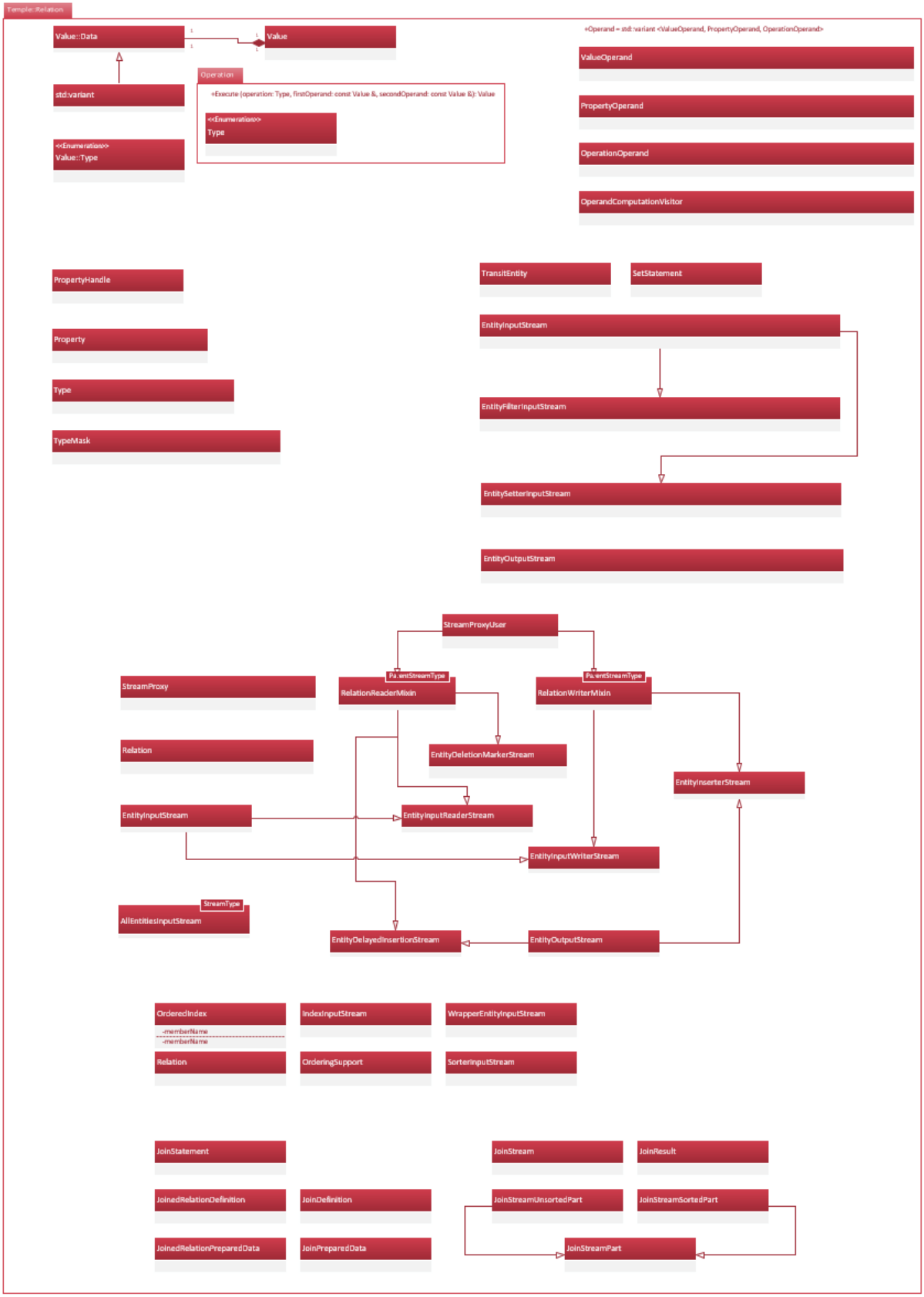
Такім чынам, хоць распрацаваная эксперыментальная база дадзеных яшчэ не гатова для выкарыстання ў камерцыйных праектах, яна паказала даволі добрыя вынікі для эксперыментальнага праекта, тым самым падцвердзіўшы перспектыўнасць яе дапрацоўкі.

ВЫСНОВЫ

У гэтай главе былі пастаўлены патрабаванні да рэалізоўваемай эксперыментальнай убудаванай базы дадзеных, арыентавай на выкарыстанне разам з шаблонам Існасць-Сістэма-Кампанент, пасля чаго былі спраектаваны і рэалізаваны наступныя часткі гэтай базы дадзеных:

- Сістэма прывязак і транзакцыённай памяці, дазваляючыя карыстальнікам базы дадзеных выкарыстоўваць прынцыпы рэактыўнага праграмавання праз даданне напісанных на C++ прывязак і валідатараў.
- Выбар базавага варыянтнага тыпа для палёў і эффектыўная рэалізацыя прымянення бінарных аперацый да варыянтных тыпаў падчас выканання праграмы.
- Сістэма метаінфармацыі для выкарыстоўваемых у картэжах тыпаў і іх палёў, дазваляючая падчас выканання звяртацца праз апісальнік да любога поля аб’екта любога выкарыстоўваемага ў картэжы базы дадзеных тыпа.
- Арганізацыя захавання дадзеных у адносінах.
- Будова формул падліку значэння і фільтрацыя картэжаў падчас выканання з дапамогай сістэмы метаінфармацыі тыпаў.
- Запыты да базы дадзеных праз плыні як універсальны інструмент перадачы і апрацоўкі дадзеных.
- Праца з плынямі ў адносінах і прымяненне мадэлі ўзаемадзеяння «чытачы-пісьменнікі» для абароны ад памылак, выклікаемых станамі гонкі.
- Сістэма індэксацыі з шчыльнымі ўпарадкаванымі індэксамі, якія могуць быць створаны ў любых адносінах па любых палях запісаў у гэтых адносінах і актуальнасць якіх будзе падтрымлівацца аўтаматычна.
- Інтэграцыя плыняў з індэксаў з сістэмай плыняў з адносінаў і механізм падрыхтоўкі інструкцыі для хуткага стварэння індэксіраванай плыні з патрэбнымі характарыстыкамі.
- Аперацыя злучэння па любой колькасці адносінаў з аўтаматычным выбарам аптымальных індэксаў і падрыхтоўкай інструкцыі для хуткага стварэння плыні злучэння.

Пасля гэтага для праверкі магчымасці практычнага выкарыстання рэалізаванай эксперыментальнай базы дадзеных і яе параўнання з бібліятэкай EnTT была напісана трывіяльная гульня. У выніку гэтай праверкі было вырашана, што хоць у распрацаванай базы дадзеных ёсць некаторыя недахопы, яна мае добрыя перспектывы для развіцця і дапрацоўкі.



Выява 3.11 – Скарочаная дыяграма класаў з усімі класамі базы дадзеных.

ЗАКЛЮЧЭННЕ

У першай главе былі разгледжаны найбольш вядомыя і выкарыстоўваемыя шаблоны праектавання гульнёвай логікі, такія як Сцэна-Вузел-Кампанент і Існасць-Кампанент-Сістэма. Пры аглядзе шаблонаў падкрэслівалася наяўнасць аналагічных механізмаў у рэляцыйных базах дадзеных, што наўпрост звязана з асноўнай ідэяй гэтай працы – стварэнне аптымізаванай пад патрабаванні гульнёвых сімуляцый убудаванай базы дадзеных можа спрасціць распрацоўку гэтых сімуляцый, замяніўшы лакальныя рэалізацыі механізмаў больш універсальнымі і праверанымі часам механізмамі з рэляцыйнай алгебры. Таксама ў канцы главы было растлумачана рашэнне ў першую чаргу арыентавацца на працу з шаблонам Існасць-Кампанент-Сістэма, які актыўна развіваецца, набірае папулярнасць і пры гэтым мае больш супадаючых з базамі дадзеных механізмаў, чым іншыя папулярныя шаблоны.

У другой главе была сабрана тэарэтычная інфармацыя, неабходная для эфектыўнай рэалізацыі індэксацыі і аперацыі злучэння ў эксперыментальнай базе дадзеных. У тым ліку былі падрабязна разгледжаны фундаментальныя тыпы індэксаў у рэляцыйных базах дадзеных. Пры аглядзе падкрэсліваліся мэты выкарыстання індэксаў, іх уплыў на фармат захавання дадзеных, выдаткаваныя на падтрымку індэксаў рэсурсы і сувязь розных тыпаў задач з тыпамі індэксаў, якія з’яўляюцца аптымальнымі рашэннямі гэтых задач. Таксама былі разгледжаны тры фундаментальныя алгарытмы выканання аперацыі злучэння: злучэнне з укладзеным цыклам, злучэнне з хэшыраваннем і зліццё. Пры гэтым былі апісаны сітуацыі, якія найлепш адпавядаюць кожнаму разглядаемаму алгарытму.

У трэцяй главе былі пастаўлены патрабаванні да эксперыментальнай базы дадзеных, выканана праектаванне і распрацоўка асноўнага функцыянала гэтай базы дадзеных. Выкананне запытаў у ёй адбываецца на аснове сістэмы плыняў дадзеных, што робіць працу з запытамі больш універсальнай і спрашчае даданне ў эксперыментальнай базу дадзеных новага функцыянала. Пры гэтым доступ да інфармацыі ў адносінах кантралюецца праз шаблон узаемадзеяння «чытачы-пісьменнікі», што дазволіць пры неабходнасці без вялікіх архітэктурных зменаў рэалізаваць магчымасць працы з адносінамі ў мнагаплынявым ассяроддзі. Таксама ў базе дадзеных была рэалізавана падтрымка шчыльных ўпарадкаваных некластэрызуючых індэксаў, у якой у тым ліку прысутнічае аўтаматычны падбор індэкса для стварэння адсартыраванай плыні з адносінаў па дадзенаму набору палёў і механізм падрыхтоўкі такіх запытаў, і рэалізацыя

аперацыі злучэння праз алгарытмы зліцця і злучэння з укладзеным цыклам, якая можа працаваць з любой колькасцю адносінаў і выкарыстоўвае механізм падрыхтоўкі запытаў.

Такім чынам галоўным вынікам гэтай працы з’яўляецца ўбудаваная база дадзеных, якую ўжо можна выкарыстоўваць як пастаўшчык існасцяў сістэмам у шаблоне Існасць-Кампанент-Сістэма ў межах эксперыментальных праектаў. Параўнанне з EnTT – вядомай рэалізацыяй шаблона Існасць-Кампанент-Сістэма з адчыненых зыходным кодам – праз распрацоўку трывіяльнай эксперыментальнай гульні падцвердзіла перспектыўнасць развіцця распрацоўваемай базы дадзеных.

СПІС ВЫКАРЫСТАНАЙ ЛІТАРАТУРЫ

1. Gregory, J. Game Engine Architecture / J. Gregory. – 3 edition – A K Peters, 2008 – 1240 pages.
2. Nystrom, R. Game Programming Patterns / R. Nystrom – 1 edition – Genever Benning, 2014 – 354 pages.
3. Overwatch Gameplay Architecture and Netcode [Электронны рэсурс] / GDC – Mode of access: https://www.youtube.com/watch?v=W3aieHjyNvw&ab_channel=GDC – Date of access: 06.12.2020.
4. What is DOTS and why is it important? [Электронны рэсурс] / Unity Learn – Mode of access: https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important – Date of access: 06.12.2020.
5. Harrington, J. L. Relational Database Design and Implementation: Clearly Explained / J. L. Harrington. – 4 edition – Morgan Kaufmann, 2016 – 712 pages.
6. Landenmaki T. Relational Database Index Design and the Optimizers / T. Landenmaki. – 1 edition – Wiley-Interscience, 2005 – 328 pages.
7. Bhattacharya A. Fundamentals of Database Indexing and Searching / A. Bhattarcharya. – 1 edition – Chapman and Hall, 2014 – 280 pages.

ДАДАТАК А

```
// Template/Relation/Structure/Operation.hpp

#pragma once

#include <Temple/Relation/Metadata/Value.hpp>
#include <Temple/Relation/Metadata/Property.hpp>

#include <boost/variant/variant.hpp>

namespace Temple::Relation
{
namespace Operation
{
enum class Type : char
{
    PLUS = 0,
    MINUS,
    MULTIPLY,
    DIVIDE,

    GREATER,
    LESS,
    EQUAL,
    NOT_EQUAL,

    AND,
    OR,

    VALUES_COUNT
};

Value Execute (Type operation, const Value &firstOperand, const Value
&secondOperand);
}
}

// Template/Relation/Structure/Operation.cpp

#include "Operation.hpp"
#include <boost/variant/get.hpp>

namespace Temple::Relation::Operation
{
template <Type operationType, Value::Type firstOperandType, Value::Type
secondOperandType>
Value Do (const Value &firstOperand, const Value &secondOperand)
{
    BOOST_ASSERT (false);
    return Value (false);
}

#define IMPLEMENT_SIMPLE_BINARY_OPERATION(OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, \
    SecondOperandTypeEnum, SecondOperandType) \
template <> Value Do <OperationType, FirstOperandTypeEnum,
SecondOperandTypeEnum> ( \
```

```
const Value &firstOperand, const Value &secondOperand) \
{ \
    return Value (boost::get <FirstOperandType> (firstOperand.GetData ())
OperationSymbol \
        boost::get <SecondOperandType> (secondOperand.GetData ())); \
}

#define
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND(Operati
onType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, Value::Type::INT32, int32_t) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, Value::Type::INT64, int64_t) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, Value::Type::FLOAT, float) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, Value::Type::DOUBLE, double) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (OperationType, OperationSymbol, \
    FirstOperandTypeEnum, FirstOperandType, Value::Type::LONG_DOUBLE, long
double)

#define IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS(OperationType,
OperationSymbol) \
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND ( \
    OperationType, OperationSymbol, Value::Type::INT32, int32_t) \
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND ( \
    OperationType, OperationSymbol, Value::Type::INT64, int64_t) \
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND ( \
    OperationType, OperationSymbol, Value::Type::FLOAT, float) \
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND ( \
    OperationType, OperationSymbol, Value::Type::DOUBLE, double) \
IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS_AS_SECOND_OPERAND ( \
    OperationType, OperationSymbol, Value::Type::LONG_DOUBLE, long double)

#define IMPLEMENT_EQUALITY_CHECK_FOR_TYPE(TypeEnum, VariableType) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::EQUAL, ==, TypeEnum, VariableType,
TypeEnum, VariableType) \
IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::NOT_EQUAL, !=, TypeEnum,
VariableType, TypeEnum, VariableType)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::PLUS, +)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::MINUS, -)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::MULTIPLY, *)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::DIVIDE, /)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::GREATER, >)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::LESS, <)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::EQUAL, ==)

IMPLEMENT_SIMPLE_BINARY_OPERATION_FOR_ALL_NUMBERS (Type::NOT_EQUAL, !=)

IMPLEMENT_EQUALITY_CHECK_FOR_TYPE (Value::Type::STRING, std::string)
```

```
IMPLEMENT_EQUALITY_CHECK_FOR_TYPE (Value::Type::BOOL, bool)

IMPLEMENT_EQUALITY_CHECK_FOR_TYPE (Value::Type::RAW_POINTER, void *)

IMPLEMENT_EQUALITY_CHECK_FOR_TYPE (Value::Type::SHARED_POINTER,
std::shared_ptr <void>)

IMPLEMENT_EQUALITY_CHECK_FOR_TYPE (Value::Type::REFERENCE, Reference)

IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::PLUS, +,
                                   Value::Type::STRING, std::string,
                                   Value::Type::STRING, std::string)

IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::GREATER, >,
                                   Value::Type::STRING, std::string,
                                   Value::Type::STRING, std::string)

IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::LESS, <,
                                   Value::Type::STRING, std::string,
                                   Value::Type::STRING, std::string)

IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::OR, ||,
                                   Value::Type::BOOL, bool,
                                   Value::Type::BOOL, bool)

IMPLEMENT_SIMPLE_BINARY_OPERATION (Type::AND, &&,
                                   Value::Type::BOOL, bool,
                                   Value::Type::BOOL, bool)

typedef Value (*OperationImplementation) (const Value &, const Value &);

#define REGISTER_SECOND_OPERAND_TYPE(Operation, FirstType, SecondType) \
    Do <Operation, FirstType, SecondType>

#define REGISTER_FIRST_OPERAND_TYPE(Operation, FirstType) \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType, Value::Type::BOOL),
\
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::INT32), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::INT64), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::FLOAT), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::DOUBLE), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::LONG_DOUBLE), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::RAW_POINTER), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::SHARED_POINTER), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::WEAK_POINTER), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::REFERENCE), \
    REGISTER_SECOND_OPERAND_TYPE (Operation, FirstType,
Value::Type::STRING)
```

```
#define REGISTER_OPERATION_TYPE(Operation) \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::BOOL), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::INT32), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::INT64), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::FLOAT), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::DOUBLE), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::LONG_DOUBLE), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::RAW_POINTER), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::SHARED_POINTER), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::WEAK_POINTER), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::REFERENCE), \
    REGISTER_FIRST_OPERAND_TYPE (Operation, Value::Type::STRING)

static OperationImplementation operationImplementationTable[
    (unsigned int) Type::VALUES_COUNT *
    (unsigned int) Value::Type::VALUES_COUNT *
    (unsigned int) Value::Type::VALUES_COUNT] =
    {
        REGISTER_OPERATION_TYPE (Type::PLUS),
        REGISTER_OPERATION_TYPE (Type::MINUS),
        REGISTER_OPERATION_TYPE (Type::MULTIPLY),
        REGISTER_OPERATION_TYPE (Type::DIVIDE),
        REGISTER_OPERATION_TYPE (Type::GREATER),
        REGISTER_OPERATION_TYPE (Type::LESS),
        REGISTER_OPERATION_TYPE (Type::EQUAL),
        REGISTER_OPERATION_TYPE (Type::NOT_EQUAL),
        REGISTER_OPERATION_TYPE (Type::AND),
        REGISTER_OPERATION_TYPE (Type::OR)
    };

Value Execute (Type operation, const Value &firstOperand, const Value
&secondOperand)
{
    return operationImplementationTable[
        (unsigned int) operation * (unsigned int) Value::Type::VALUES_COUNT
*
        (unsigned int) Value::Type::VALUES_COUNT +
        (unsigned int) firstOperand.GetType () * (unsigned int)
Value::Type::VALUES_COUNT +
        (unsigned int) secondOperand.GetType ()] (firstOperand,
secondOperand);
}
}
```

ДАДАТАК Б

```
// Template/Relation/Metadata/Type.hpp

#pragma once
#include <Temple/Relation/Metadata/Property.hpp>
#include <vector>

#include <boost/dynamic_bitset.hpp>

namespace Temple::Relation
{
class Type final
{
public:
    Type ();
    ~Type () = default;

    void RegisterProperty (Property property, PropertyHandle
&registeredHandle);
    const std::vector <Property> &GetProperties () const;

    bool operator== (const Type &other) const;
    bool operator!= (const Type &other) const;

private:
    std::vector <Property> properties_;
};

class TypeMask final
{
public:
    explicit TypeMask (const Type &source);
    ~TypeMask () = default;

    const Type *GetSource () const;
    void Exclude (const PropertyHandle &propertyHandle);
    bool IsRedundant () const;
    const Property *GetProperty (const PropertyHandle &propertyHandle)
const;

private:
    inline void CheckPropertyHandle (const PropertyHandle &propertyHandle)
const;

    const Type *source_;
    boost::dynamic_bitset <> propertyAvailabilityMask_;
};
}

// Template/Relation/Metadata/Type.cpp

#include "Type.hpp"

#include <boost/assert.hpp>

namespace Temple::Relation
{
```

```
Type::Type ()
    : properties_ ()
{
}

void Type::RegisterProperty (Property property, PropertyHandle
&registeredHandle)
{
    registeredHandle.type_ = this;
    registeredHandle.index_ = properties_.size ();
    properties_.push_back (std::move (property));
}

const std::vector <Property> &Type::GetProperties () const
{
    return properties_;
}

bool Type::operator == (const Type &other) const
{
    return this == &other;
}

bool Type::operator != (const Type &other) const
{
    return !(other == *this);
}

TypeMask::TypeMask (const Type &source)
    : source_ (&source),
      propertyAvailabilityMask_ (source.GetProperties ().size ())
{
    propertyAvailabilityMask_.set ();
}

const Type *TypeMask::GetSource () const
{
    return source_;
}

void TypeMask::Exclude (const PropertyHandle &propertyHandle)
{
    CheckPropertyHandle (propertyHandle);
    propertyAvailabilityMask_[propertyHandle.index_] = false;
}

bool TypeMask::IsRedundant () const
{
    return propertyAvailabilityMask_.none ();
}

const Property *TypeMask::GetProperty (const PropertyHandle
&propertyHandle) const
{
    CheckPropertyHandle (propertyHandle);
    if (propertyAvailabilityMask_[propertyHandle.index_])
    {
        return &source_->GetProperties ()[propertyHandle.index_];
    }
```

```
    else
    {
        return nullptr;
    }
}

void TypeMask::CheckPropertyHandle (const PropertyHandle &propertyHandle)
const
{
    BOOST_ASSERT (*propertyHandle.type_ == *source_);
    BOOST_ASSERT (propertyHandle.index_ < source_->GetProperties ().size
());
}
}

// Template/Relation/Metadata/Property.hpp

#pragma once

#include <functional>
#include <Temple/Relation/Metadata/Value.hpp>
#include <Temple/Transaction/Transaction.hpp>

namespace Temple::Relation
{
class Type;

struct PropertyHandle
{
    const Type *type_;
    std::size_t index_;
};

class Property final
{
public:
    typedef std::function <Value (void *)> ValueGetter;
    typedef std::function <Transaction::Transaction <Value> & (void *)>
TransactionGetter;

    Property (Value::Type type, bool transient,
              ValueGetter valueGetter, TransactionGetter
transactionGetter);

    ~Property () = default;

    Value::Type GetType () const;

    bool IsTransient () const;

    Value Get (void *entityInstance) const;

    Transaction::Transaction <Value> &Set (void *entityInstance) const;

private:
    Value::Type type_;
    bool transient_;

    ValueGetter valueGetter_;
```

```
    TransactionGetter transactionGetter_;
};
}

// Template/Relation/Metadata/Property.cpp

#include "Property.hpp"

#include <utility>

namespace Temple::Relation
{
Property::Property (Value::Type type, bool transient,
                    Property::ValueGetter valueGetter,
Property::TransactionGetter transactionGetter)
    : type_ (type),
      transient_ (transient),
      valueGetter_ (std::move (valueGetter)),
      transactionGetter_ (std::move (transactionGetter))
{
}

Value::Type Property::GetType () const
{
    return type_;
}

bool Property::IsTransient () const
{
    return transient_;
}

Value Property::Get (void *entityInstance) const
{
    return valueGetter_ (entityInstance);
}

Transaction::Transaction <Value> &Property::Set (void *entityInstance)
const
{
    return transactionGetter_ (entityInstance);
}
}
```

ДАДАТАК В

```
// Template/Relation/Structure/Operand.hpp

#pragma once
#include <Temple/Relation/Structure/Operation.hpp>
#include <Temple/Relation/Structure/Entity.hpp>

namespace Temple::Relation
{
class ValueOperand final
{
public:
    explicit ValueOperand (const Value &value);
    ~ValueOperand () = default;
    const Value &GetValue () const;

private:
    Value value_;
};

class PropertyOperand final
{
public:
    explicit PropertyOperand (const PropertyHandle &handle);
    ~PropertyOperand () = default;

    const PropertyHandle &GetPropertyHandle () const;
    Value ComputeValue (const Header &header, const Entity &entity) const;

private:
    PropertyHandle handle_;
};

class OperationOperand;
using Operand = boost::variant <ValueOperand, PropertyOperand,
OperationOperand>;

class OperationOperand final
{
public:
    OperationOperand (Operation::Type operation, Operand first, Operand
second);
    OperationOperand (const OperationOperand &other);
    OperationOperand (OperationOperand &&other);
    ~OperationOperand ();

    Operation::Type GetOperation () const;
    const Operand &GetFirst () const;
    const Operand &GetSecond () const;
    Value ComputeValue (const Header &header, const Entity &entity) const;

    OperationOperand &operator =(const OperationOperand &other);

private:
    Operation::Type operation_;
    Operand *operands_;
};
```

```
class OperandComputationVisitor final : public boost::static_visitor
<Value>
{
public:
    OperandComputationVisitor (const Header &header, const Entity &entity);
    ~OperandComputationVisitor () = default;

    Value operator() (const ValueOperand &operand) const;
    Value operator() (const PropertyOperand &operand) const;
    Value operator() (const OperationOperand &operand) const;

private:
    const Header &header_;
    const Entity &entity_;
};
}

// Template/Relation/Structure/Operand.cpp

#include "Operand.hpp"

namespace Temple::Relation
{
ValueOperand::ValueOperand (const Value &value)
    : value_ (value)
{
}

const Value &ValueOperand::GetValue () const
{
    return value_;
}

PropertyOperand::PropertyOperand (const PropertyHandle &handle)
    : handle_ (handle)
{
}

const PropertyHandle &PropertyOperand::GetPropertyHandle () const
{
    return handle_;
}

Value PropertyOperand::ComputeValue (const Header &header, const Entity
&entity) const
{
    if (*header.GetSource () == *handle_.type_)
    {
        const Property *property = header.GetProperty (handle_);
        BOOST_ASSERT (property);
        return property->Get (entity.get ());
    }

    BOOST_ASSERT (false);
    return Value (false);
}
```

```
OperationOperand::OperationOperand (Operation::Type operation, Operand
first, Operand second)
    : operation_ (operation),
      operands_ ((Operand *) malloc (sizeof (Operand) * 2))
{
    new (operands_) Operand (std::move (first));
    new (operands_ + 1) Operand (std::move (second));
}

OperationOperand::OperationOperand (const OperationOperand &other)
    : operation_ (other.operation_),
      operands_ ((Operand *) malloc (sizeof (Operand) * 2))
{
    new (operands_) Operand (other.GetFirst ());
    new (operands_ + 1) Operand (other.GetSecond ());
}

OperationOperand::OperationOperand (OperationOperand &&other)
    : operation_ (other.operation_),
      operands_ (other.operands_)
{
    other.operands_ = nullptr;
}

OperationOperand::~OperationOperand ()
{
    if (operands_)
    {
        operands_[0].~Operand ();
        operands_[1].~Operand ();
        free (operands_);
    }
}

Operation::Type OperationOperand::GetOperation () const
{
    return operation_;
}

const Operand &OperationOperand::GetFirst () const
{
    return operands_[0];
}

const Operand &OperationOperand::GetSecond () const
{
    return operands_[1];
}

Value OperationOperand::ComputeValue (const Header &header, const Entity
&entity) const
{
    Value firstOperand = boost::apply_visitor (OperandComputationVisitor
(header, entity), GetFirst ());
    Value secondOperand = boost::apply_visitor (OperandComputationVisitor
(header, entity), GetSecond ());
    return Operation::Execute (operation_, firstOperand, secondOperand);
}
```

```
OperationOperand &OperationOperand::operator = (const OperationOperand
&other)
{
    if (&other == this)
    {
        return *this;
    }

    operation_ = other.operation_;
    operands_[0] = other.GetFirst ();
    operands_[1] = other.GetSecond ();
    return *this;
}

OperandComputationVisitor::OperandComputationVisitor (const Header &header,
const Entity &entity)
    : header_ (header),
      entity_ (entity)
{}

Value OperandComputationVisitor::operator () (const ValueOperand &operand)
const
{
    return operand.GetValue ();
}

Value OperandComputationVisitor::operator () (const PropertyOperand
&operand) const
{
    return operand.ComputeValue (header_, entity_);
}

Value OperandComputationVisitor::operator () (const OperationOperand
&operand) const
{
    return operand.ComputeValue (header_, entity_);
}
}
```