

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



Sviluppo di un'estensione per la Transparent
Data Encryption in PostgreSQL

Tesi di laurea

Relatore

Prof. Michele Scquizzato

Laureando

Filippo Venzo

Matricola 2113705

Sommario

Il presente documento descrive il lavoro svolto dal laureando Filippo Venzo durante il periodo di stage presso l'azienda Miriade Srl. Lo scopo è illustrare il contesto aziendale, gli obiettivi e le attività svolte, fornendo infine una valutazione sul lavoro compiuto.

Lo stage è stato svolto al termine del percorso di studi della laurea triennale in Informatica per una durata di trecentoquattro ore e il suo fine è stato quello di sviluppare un'estensione di PostgreSQL per l'adozione della *Transparent Data Encryption*.

Indice

1	Introduzione	1
1.1	Contesto aziendale	1
1.2	Progetto	2
1.2.1	Motivazioni	2
1.2.2	Obiettivo	2
1.2.3	Stato dell'arte	2
1.2.4	Attività preventivate	3
1.3	Metodo di lavoro	4
1.4	Tecnologie utilizzate	4
1.4.1	C	4
1.4.2	PostgreSQL	5
1.4.3	OpenSSL	5
1.4.4	Perl	5
1.5	Strumenti utilizzati	6
1.5.1	make	6
1.5.2	Bitbucket	6
1.5.3	Jira	6
1.5.4	Confluence	6
1.6	Organizzazione del testo	6
2	Specifiche software	9
2.1	Stato del prodotto	9
2.2	Funzionalità pregresse	10
2.2.1	Cifratura delle tabelle	10
2.2.2	Cifratura degli indici	12
2.3	Gestione delle chiavi crittografiche	13
2.3.1	DEK	13
2.3.2	KEK	15
2.4	Configurazione tramite parametri GUC	16
3	Sviluppo	19
3.1	Crittografia delle tabelle	19

3.1.1	Flusso di inserimento <i>TOAST</i>	19
3.1.2	Soluzione definitiva	20
3.1.3	CLUSTER e VACUUM FULL	22
3.2	Utility di backup logico	23
3.2.1	Architettura	24
3.2.2	Sicurezza del backup	25
3.2.3	Test e Validazione	26
3.3	Crittografia degli indici	26
3.3.1	<i>Operator class</i> e <i>operator family</i>	27
3.3.2	AES-256-SIV	28
3.3.3	Architettura di <i>tde_btree</i>	29
3.3.4	Limitazioni	30
3.4	Auditing	30
3.4.1	Formato dei messaggi	31
3.4.2	Eventi registrati	31
3.5	Attività di manutenzione	32
3.5.1	Rotazione delle chiavi	32
3.5.2	Rimozione della DEK globale	35
3.5.3	Ottimizzazione delle performance	35
3.5.4	Sistema di configurazione	38
3.5.5	Modifica al formato cifrato	38
4	Verifica	41
4.1	Regressione	41
4.2	Isolamento	43
4.3	TAP	43
4.4	Performance	44
5	Conclusioni	45
5.1	Limitazioni	45
5.2	Valutazione personale	46
	Acronimi e abbreviazioni	47
	Glossario	48
	Bibliografia	50

Capitolo 1

Introduzione

1.1 Contesto aziendale

Miriade Srl è una società di consulenza informatica fondata nel 2000, con sede principale a Thiene (VI) e una filiale a Padova.¹ L'offerta si concentra principalmente su soluzioni e servizi per creare ecosistemi efficienti di dati, operando in ambiti quali il Data Management (Database, Data Integration, Data Governance), le Analytics (Business Intelligence, Machine Learning, Process Automation) ed Infrastructure & Operations. Inoltre, si occupa dello sviluppo di applicativi (*Modern Application*), con un focus particolare sulla progettazione di API, architetture a microservizi e soluzioni per la Business Continuity. In qualità di *AWS Advanced Consulting Partner*, Miriade vanta consolidate competenze nel Cloud Adoption, Cloud Native Development e pratiche di DevOps.

Attualmente l'azienda è composta da un organico di circa 100 dipendenti e opera in vari settori come la manifattura, la pubblica amministrazione e i servizi.

L'azienda è suddivisa in diverse aree come *Analytics*, *IT&Operations*, *Applications* e *Data*. Il tirocinio si è inserito all'interno di quest'ultima area, dove sono presenti figure professionali come *Database Administrator* e *Data Architect*.



Figura 1.1: Il logo di Miriade Srl

¹ *Miriade Srl - Sito Ufficiale*. URL: <https://www.miriade.it>.

1.2 Progetto

1.2.1 Motivazioni

La protezione dei dati "*at rest*", ovvero tutti i dati sensibili conservati in supporti non volatili, è una delle norme stabilite dal [General Data Protection Regulation \(GDPR\)](#),² che richiede la pseudonimizzazione e la cifratura dei dati personali. La conformità a tale norma può essere raggiunta attraverso la [Transparent Data Encryption \(TDE\)](#), ovvero la crittazione dei file del *database* che risiedono sul disco fisso, detti dati inattivi, i quali vengono decifrati in memoria volatile durante le operazioni di I/O in quanto assumono lo stato di dati in transito. Tale processo è detto trasparente poiché non impatta in alcun modo utenti e applicazioni autorizzati nell'utilizzo della base di dati, ma offre un efficace livello di sicurezza contro l'accesso non consentito ai file fisici.

La cifratura è il procedimento di trasformazione di testo in chiaro, leggibile ed interpretabile da un umano, in un testo cifrato e illeggibile, composto da sequenze casuali di caratteri prodotte da un algoritmo.

1.2.2 Obiettivo

Il progetto di Miriade nasce dalla mancanza della [TDE](#) nativa all'interno di PostgreSQL, il che costringe molte aziende, tra cui Miriade stessa, a ricorrere a soluzioni invasive come la cifratura degli interi supporti di memorizzazione. L'obiettivo quindi è stato proprio quello di realizzare `pg_vault_tde`, un'estensione per la [TDE](#) in PostgreSQL sfruttando le [Application Programming Interface \(API\)](#) di estensibilità di quest'ultimo. I requisiti principali erano quelli di poter cifrare le tabelle e l'intero spettro degli indici senza modificare il *core* del database. Inoltre di fondamentale importanza era la delegazione della gestione delle chiavi di crittografia a dei [Key Management System \(KMS\)](#) esterni come HashiCorp Vault e OpenBao.

1.2.3 Stato dell'arte

Differentemente da altri [Database Management System \(DBMS\)](#) come Oracle Database, Microsoft SQL Server, MySQL e MariaDB, PostgreSQL nativamente non supporta la [TDE](#). Si tratta di una scelta deliberata dei suoi sviluppatori, frutto di numerose discussioni anche attuali,³ motivata da precise ragioni. Non è chiaro quale *threat-model*, ovvero modello di minaccia alla sicurezza, la [TDE](#) possa risolvere. Se la minaccia fosse il furto del disco fisico si potrebbe intervenire sulla cifratura dell'intero disco; se lo fosse invece un amministratore di sistema malintenzionato, allora la *transparent encryption* si rivelerebbe inutile. Il problema che la [TDE](#) potrebbe maggiormente contribuire a risolvere sarebbe quello della *compliance*,

²Unione Europea. *Regolamento (UE) 2016/679 del Parlamento europeo e del Consiglio del 27 aprile 2016 (GDPR)*. in *GUUE* L 119 del 4 maggio 2016. Art. 32, par. 1, lett. a). 2016.

³Insung Moon. *[Proposal] Table-level Transparent Data Encryption (TDE) and Key Management Service (KMS)*. 2018. URL: [https://www.postgresql.org/message-id/flat/031401d3f41d\\$5c70ed90\\$1552c8b0\\$@lab.ntt.co.jp](https://www.postgresql.org/message-id/flat/031401d3f41d$5c70ed90$1552c8b0$@lab.ntt.co.jp); Peter Eisentraut. *Transparent column encryption*. 2021. URL: <https://www.postgresql.org/message-id/flat/89157929-c2b6-817b-6025-8e4b2d89d88f@enterprisedb.com>.

ovvero l'aderenza delle aziende alle norme in tema di protezione dei dati (GDPR, DSS, HIPAA). Proprio quest'ultimo punto non è ritenuto sufficientemente valido per giustificare la complessità tecnica dell'introduzione della **TDE** in PostgreSQL e l'*overhead* che sarebbe prodotto da quest'ultima.

Nonostante questo, sono nati alcuni progetti per l'introduzione di questo tipo di crittografia nel **DBMS**, accomunati tutti dal problema di aver fatto un *fork* del codice sorgente di PostgreSQL e apportato le proprie modifiche. Questo costringe l'utilizzatore ad un *vendor lock-in*, ovvero a dipendere non più dalla versione ufficiale di PostgreSQL ma da quella mantenuta dal fornitore della **TDE**.

L'obiettivo di **pg_vault_tde** è anche quello di non modificare in alcun modo il *core* ma adottare un approccio basato sugli *hook* e le **API** già fornite.

L'implementazione esistente della **TDE** in PostgreSQL che maggiormente si avvicina a quanto perseguito dal progetto di tirocinio è **pg_tde** dell'azienda statunitense *Percona*.⁴

1.2.4 Attività preventive

Parte dell'estensione era già stata sviluppata prima dell'inizio dello stage; infatti, le attività di sviluppo sono state suddivise in due gruppi. Il primo comprende le attività di manutenzione e risoluzione dei bug nel codice preesistente, mentre il secondo riguarda le attività di progettazione e sviluppo di nuove funzionalità per **pg_vault_tde**.

Il piano di lavoro ha previsto le seguenti attività settimanali:

- **Settimana 1:** formazione su PostgreSQL come **DBMS**, attraverso la visione di contenuti elaborati da Miriade Srl, lo studio della documentazione di PostgreSQL, in particolare del capitolo 36⁵ che tratta la costruzione di estensioni, e infine l'analisi del codice presente.

Contestualmente allo studio, è stato previsto lo svolgimento di alcuni esercizi su PostgreSQL con la finalità di familiarizzare con esso.

- **Settimane 2-3:** progettazione, sulla base delle componenti esistenti, e sviluppo della cifratura delle tabelle *TOAST*. La mancanza di questa funzionalità, infatti, era un limite noto dell'estensione. È stata data particolare importanza al design di un metodo che permettesse di ottenere il minimo *overhead* durante la scrittura e la lettura di tali tabelle.

All'implementazione è seguita la fase di *testing*.

- **Settimane 4-5:** progettazione ed implementazione di **pg_dump_tde**, *utility* di *logical backup* criptato, e **pg_restore_tde**, *utility* che permette di ripristinare i backup creati dal precedente strumento.

All'implementazione è seguita la fase di *testing*.

⁴Repository *pg_tde* di Percona. URL: https://github.com/percona/pg_tde.

⁵Chapter 36: *Extending SQL*. URL: <https://www.postgresql.org/docs/18/extend.html>.

Le attività di sviluppo dovevano essere precedute da un paio di giorni dedicati allo studio del funzionamento delle analoghe *utility* `pg_dump` e `pg_restore`, fornite da PostgreSQL, che lavorano però con dati in chiaro.

- **Settimane 6-7-8:** progettazione, sulla base delle componenti esistenti, e sviluppo dell'indice `tde_btree`, basato sul normale `btree` ma con *Index Key* criptate.

Anche questo era un limite, in quanto le chiavi con tipo a lunghezza fissa (int, date, varchar, ecc.) erano scritte completamente in chiaro.

All'implementazione è seguita la fase di *testing*.

Anche in questo caso, l'*overhead* che la crittografia poteva aggiungere era cruciale.

1.3 Metodo di lavoro

La metodologia di gestione di progetto adottata durante il tirocinio è stata quella del lavoro Agile e il suo framework Scrum. In particolare i concetti del *framework* applicati sono stati i seguenti:

- Sprint: periodo temporale di durata prefissata all'interno del quale viene realizzato un incremento significativo dello stato di prodotto;
- Sprint Planning: cerimonia durante la quale viene stabilito lo Sprint Goal (obiettivo dello sprint) e lo Sprint Backlog (insieme di attività da completare durante lo sprint). Svolta ogni primo giorno dello Sprint;
- Sprint Review: cerimonia svolta al termine dello sprint, volta a mostrare l'incremento di prodotto agli *stakeholder* e raccogliere un riscontro da essi. Svolta ogni ultimo giorno dello Sprint;
- Sprint Retrospective: cerimonia svolta dopo la Sprint Review in cui il team analizza il metodo di lavoro adottato per individuarne eventuali miglioramenti possibili.

1.4 Tecnologie utilizzate

1.4.1 C

Per la realizzazione del codice dell'estensione è stato scelto il linguaggio C, principalmente perché è il linguaggio in cui il *core* di PostgreSQL è scritto; inoltre, la stessa documentazione lo indica come il migliore per la scrittura di nuove funzionalità per il DBMS,⁶ in quanto, a differenza di linguaggi come PL/pgSQL, PL/Python e PL/Perl, non è interpretato ma compilato come libreria condivisa, caricata dinamicamente dal server PostgreSQL.

⁶Documentazione PostgreSQL. URL: <https://www.postgresql.org/docs/current/xfunc-c.html>.

1.4.2 PostgreSQL

PostgreSQL⁷ è un sistema di gestione di database relazionali ad oggetti (ORDBMS) open source. Ha oltre 35 anni di storia di sviluppo; la prima versione con supporto nativo del linguaggio SQL è stata rilasciata nel 1996. A differenza di altri DBMS molto diffusi, come MySQL o Oracle Database, PostgreSQL implementa il modello *object-relational*: unisce il modello relazionale con il concetto di oggetti, consentendo il supporto di tipi di dati complessi e personalizzati, l'ereditarietà tra tabelle e l'esecuzione di funzioni direttamente nel database.

È possibile aggiungere funzionalità a PostgreSQL attraverso le API di estensibilità.

Grazie alla sua stabilità, affidabilità e velocità, è uno dei DBMS più utilizzati al mondo, adottato in produzione da aziende come Apple, Spotify e NASA. Inoltre, è una tecnologia totalmente gratuita supportata da una vasta community globale.

1.4.3 OpenSSL

OpenSSL è una libreria open source: la componente `libssl` fornisce implementazioni dei protocolli TLS (*Transport Layer Security*) e SSL (*Secure Sockets Layer*), mentre `libcrypto` offre implementazioni di algoritmi crittografici a chiave simmetrica e asimmetrica. OpenSSL è una delle librerie più diffuse per la sicurezza delle comunicazioni di rete.

In particolare, nel progetto `pg_vault_tde` è stata utilizzata l'implementazione dell'algoritmo AES-256-GCM.⁸ AES (*Advanced Encryption Standard*) è lo standard per la cifratura a blocchi su cui si basa l'algoritmo; 256 indica la lunghezza della chiave in bit; GCM (*Galois/Counter Mode*) è una modalità di funzionamento che fornisce cifratura autenticata in modo da garantire l'integrità e l'autenticità del testo cifrato mediante il calcolo di un tag di autenticazione.

È stato scelto per l'equilibrio tra sicurezza e prestazioni: è ampiamente riconosciuto come sicuro se usato correttamente, ottimizzato dall'hardware moderno e adatto a implementazioni parallele.

1.4.4 Perl

Perl è un linguaggio di programmazione ad alto livello, interpretato, nato per la manipolazione ed elaborazione di testi. All'interno del progetto `pg_vault_tde`, è stato utilizzato per la scrittura e l'esecuzione dei test di integrazione ed *end-to-end* Test Anything Protocol (TAP).⁹ È stato scelto in quanto rappresenta lo strumento ufficiale utilizzato da PostgreSQL stesso.

⁷Sito PostgreSQL. URL: <https://www.postgresql.org/>.

⁸David A. McGrew e John Viega. *The Security and Performance of the Galois/Counter Mode of Operation (Full Version)*. Cryptology ePrint Archive, Paper 2004/193. 2004. URL: <https://eprint.iacr.org/2004/193>.

⁹Documentazione PostgreSQL TAP Tests. URL: <https://www.postgresql.org/docs/current/regress-tap.html>.

1.5 Strumenti utilizzati

1.5.1 make

make è uno strumento di automazione della *build* che permette la gestione della fase di compilazione e di *linking* dei file di progetti software complessi. Nel progetto di tirocinio è stato utilizzato con l'infrastruttura di compilazione PGXS, fornita da PostgreSQL per la costruzione di estensioni.

1.5.2 Bitbucket

Il versionamento del codice è stato fatto tramite *git* e con Atlassian Bitbucket come servizio di hosting. Quest'ultimo è infatti una piattaforma di gestione di repository *git* che permette il controllo di versione attraverso un'interfaccia grafica. Il *workflow* adottato durante il progetto è stato il *Gitflow*¹⁰, che prevede lo sviluppo delle funzionalità in rami dedicati, al cui termine vengono riuniti con il ramo di sviluppo principale, che a sua volta, quando è pronto, viene unito nel ramo di produzione.

1.5.3 Jira

Come **Issue Tracking System (ITS)** è stato scelto Jira, in quanto, facendo parte della suite Atlassian, permette un'integrazione completa con Bitbucket. Inoltre, attraverso l'organizzazione in Sprint e Kanban, offre la possibilità di gestire le metodologie di sviluppo Agile, distribuendo le attività nei Backlog dei vari periodi.

1.5.4 Confluence

Confluence, della suite Atlassian, è una piattaforma di *Knowledge Management* e *Wiki*, che permette la centralizzazione e la condivisione della documentazione tecnica ed organizzativa. Nel contesto del progetto di tirocinio è stato utilizzato per la redazione dei verbali delle cerimonie di Agile/Scrum, report di bug e illustrazione di flussi computazionali complessi.

1.6 Organizzazione del testo

Il **secondo capitolo** descrive lo stato iniziale del prodotto, illustrando l'architettura preesistente e i requisiti software da soddisfare;

Il **terzo capitolo** approfondisce il ciclo di sviluppo del progetto, dettagliando le scelte progettuali e le soluzioni implementative adottate;

Il **quarto capitolo** espone l'architettura di collaudo e le diverse tipologie di test eseguiti per validare le funzionalità e misurarne le prestazioni;

¹⁰ *Gitflow*. URL: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>.

Il quinto capitolo illustra il consuntivo finale del tirocinio, con un bilancio sugli obiettivi raggiunti e una valutazione sulle competenze acquisite.

Riguardo alla stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- i termini in lingua straniera o facenti parte del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Specifiche software

2.1 Stato del prodotto

Come precedentemente descritto, una parte del progetto `pg_vault_tde` è stata sviluppata prima dell'inizio del tirocinio. Il codice preesistente, scritto in linguaggio C, includeva i seguenti moduli:

- Un sistema di cifratura delle tuple basato sull'algoritmo AES-256-GCM.
- Un sistema di cifratura delle chiavi degli indici basato su AES-256-SIV, un algoritmo di cifratura deterministico. Questa scelta è motivata dalla necessità di supportare l'operatore di uguaglianza (`=`), che richiede la generazione del medesimo testo cifrato (*ciphertext*) a partire dallo stesso testo in chiaro (*plaintext*). Tuttavia, tale componente non era inizialmente funzionante;
- *encrypted_heap*: un [Table Access Method \(TAM\)](#) personalizzato in sostituzione del predefinito di PostgreSQL, denominato `heap`. Il compito di *encrypted_heap* consiste nel cifrare i dati prima della scrittura su disco, la quale viene poi demandata al sottosistema di archiviazione nativo del database. Il modulo risultava operativo per le sole tuple contenenti colonne a lunghezza fissa, mentre presentava limitazioni con attributi con tipo a lunghezza variabile che superavano la soglia dei 2 KB, i quali richiedono il meccanismo [The Oversized-Attribute Storage Technique \(TOAST\)](#);
- *tde_btree*: un [Index Access Method \(IAM\)](#) personalizzato, derivato dall'indice `btree` standard di PostgreSQL, deputato alla cifratura delle chiavi di indicizzazione (*Index Key*), ossia le colonne oggetto dell'indice stesso. Tale componente era in grado di cifrare unicamente colonne con tipi di dato a lunghezza variabile, mentre la fase di accesso all'indice e di recupero dei dati non era funzionante;
- Un sistema di gestione delle chiavi crittografiche [Key Encryption Key \(KEK\)](#) e una tabella di catalogo per la memorizzazione e la gestione delle chiavi di cifratura dei dati ([Data Encryption Key \(DEK\)](#)).

L'adozione di una gerarchia a due livelli è necessaria per motivi di sicurezza: l'utilizzo di una singola chiave **DEK** salvata all'interno dello stesso database esporrebbe il sistema al rischio che un potenziale attaccante, ottenendo l'accesso ai file di dati, possa recuperare anche la chiave necessaria per decifrarli. Di conseguenza, tutte le **DEK** memorizzate nel database vengono cifrate tramite la **KEK**, che risiede esternamente al database nel **KMS**;

- Un sistema di *caching* delle **DEK** volto ad evitare di leggere dalla tabella di catalogo ad ogni operazione crittografica.
- Un sistema di configurazione dell'estensione basato sul sistema GUC¹ di PostgreSQL.

2.2 Funzionalità pregresse

I moduli precedentemente descritti offrivano le seguenti funzionalità:

- Creazione di tabelle cifrate (prive di colonne a lunghezza variabile);
- Lettura e interrogazione di tabelle cifrate;
- Eliminazione di tabelle cifrate;
- Configurazione di un **KMS** remoto tramite HashiCorp Vault;
- Configurazione di un **KMS** locale basato sullo standard PKCS#12.

2.2.1 Cifratura delle tabelle

La cifratura delle tabelle viene eseguita a livello di tupla: l'entità minima elaborata dall'algoritmo di cifratura è, pertanto, la singola riga. Ogni riga viene cifrata utilizzando la chiave **DEK** associata alla tabella di appartenenza, la quale viene memorizzata all'interno della tabella di catalogo `pg_vault_tde_catalog`, gestita direttamente dall'estensione.

L'implementazione è stata resa possibile grazie all'architettura di PostgreSQL, che espone il **TAM** e le relative funzioni di *callback*. Si tratta di puntatori a funzione che possono essere modificati, all'avvio del server, affinché facciano riferimento alle funzioni personalizzate definite dall'utente nella struttura `TableAmRoutine`, che definisce il nuovo **TAM**.

Le funzioni di *callback* più significative esposte dal **TAM** per le finalità di `pg_vault_tde` sono:

- `tuple_insert()` richiamata ogniqualvolta avvenga l'inserimento di una riga in una tabella;
- `tuple_update()` richiamata ogniqualvolta avvenga la modifica di una o più tuple;

¹Documentazione PostgreSQL: Grand Unified Configuration (GUC). URL: <https://www.postgresql.org/docs/current/config-setting.html>.

- `scan_getnextslot()` richiamata ogniqualvolta il sistema debba estrarre una o più tuple da una tabella (ad esempio, durante l'esecuzione di una query `SELECT`), durante uno *sequential scan*.

La ridefinizione di tali funzioni consente di:

- cifrare le tuple prima dell'effettivo inserimento su disco, delegando poi la scrittura fisica alla funzione `heap_insert()` del `TAM` di default (*heap*);
- cifrare la nuova versione della tupla e decifrare la precedente prima dell'effettivo aggiornamento sul disco, delegando poi l'aggiornamento alla funzione `heap_update()`;
- decifrare la tupla a seguito del suo recupero dalla tabella, prima che venga restituita all'utente o ai livelli superiori del DBMS.

L'introduzione del `TAM encrypted_heap` risulta pertanto fondamentale per raggruppare ed eseguire le nuove funzioni di cifratura di `pg_vault_tde`.

Al di fuori delle operazioni di cifratura, le attività di manipolazione delle tuple non sono state reimplementate da zero, ma si è scelto invece di riutilizzare quelle native di *heap* fornite da PostgreSQL. La progettazione, lo sviluppo e la manutenzione di metodi alternativi ad ogni rilascio del DBMS avrebbero infatti comportato una complessità e un dispendio di risorse eccessivi. Il riuso è reso possibile da `saved_am`, ossia una copia di *heap* utilizzata per sovrascrivere tutte le *callback* della struttura `TableAmRoutine` non influenzate dalla cifratura.

Formato della tupla cifrata

Ogni tupla all'interno di PostgreSQL è composta da due elementi: un *header* contenente le informazioni strutturali della tupla come il numero di attributi, il loro tipo e la loro lunghezza, e un *payload* che rappresenta i dati. Ciò che viene cifrato da `pg_vault_tde` è solamente il contenuto informativo; l'intestazione rimane in chiaro. Il formato del contenuto cifrato era:

```
[ VERSION_BYTE(1) | GENERATION(8) | IV(12) | CT(N) | TAG(16) ]
```

dove l'unità di misura sottintesa è il byte ed i campi sono:

- *Version byte*, byte di versione del formato cifrato in previsione di future modifiche. Permette di identificare tuple cifrate con formati precedenti e mantenere la compatibilità con essi;
- *Generation*, generazione della chiave con cui è stata cifrata la tupla;
- *IV*, è l'*Initialization Vector (IV)* univoco utilizzato in fase di cifratura;
- *CT*, è il testo cifrato (in inglese *ciphertext*);
- *TAG*, è il campo calcolato da GCM per garantire integrità e autenticazione del dato.

Limite attributi *TOASTed*

Come sopracitato, all'inizio del progetto la funzionalità di cifratura presentava alcuni limiti, quali l'impossibilità di cifrare tuple con colonne a lunghezza variabile, come quelle di tipo TEXT o BYTEA.

Questo era dovuto al modo in cui PostgreSQL organizza i dati memorizzati sul disco, ovvero in pagine da 8 KB, e impedisce a una riga di espandersi in più pagine di memoria. Proprio per questo, agli attributi con una lunghezza maggiore di 2 KB viene applicato il meccanismo **TOAST**, il quale prima tenta di comprimere il dato in questione e, se questo supera ancora la soglia dei 2 KB, lo sposta *out-of-line*, ovvero in una tabella esterna detta *toast table*, all'interno della quale viene diviso in diverse tuple chiamate *chunk*.

L'applicazione di tale meccanismo avviene automaticamente all'interno della funzione `heap_toast_insert_or_update()`, richiamata proprio per questo da `pg_vault_tde` sulle tuple ancora in chiaro. Questo accadeva per una scelta implementativa basata sull'assunzione errata che la *toast table* ereditasse automaticamente l'*access method* dalla tabella genitore e che quindi, prima del loro inserimento, i *chunk* venissero crittografati. Un'analisi del codice sorgente di PostgreSQL ha rivelato che il metodo sopracitato, appartenente a *heap*, bypassa completamente le funzioni definite dal nuovo **TAM** ed inserisce i *chunk* non cifrati attraverso la funzione `heap_insert()`.

Contrariamente, per la lettura di tutte le tuple cifrate, anche per quelle con attributi *toast*, vengono applicati i metodi definiti da `encrypted_heap` e quindi veniva tentata la decifratura su testi in chiaro, provocando errori e perdita di dati.

2.2.2 Cifratura degli indici

La cifratura degli indici, in particolare delle *Index Key*, avviene tramite la creazione di un nuovo **IAM**: `tde_btree`. La sua struttura è analoga a quella di `encrypted_heap`: è presente una *struct* denominata `IndexAmRoutine` che espone le *callback* sovrascrivibili.

`tde_btree` parte da una copia dell'**IAM** dell'indice *btree* di PostgreSQL e sovrascrive solo le funzioni che sono influenzate dalla cifratura; alcune di queste sono:

- `*ambuild()` richiamata ogniqualvolta viene costruito un indice del tipo specificato. La cifratura è gestita da `tde_btree` mentre la costruzione è demandata ai metodi di *btree*.
- `*aminsert()` richiamata ogniqualvolta viene inserita una tupla con colonna indicizzata. Anche qui la costruzione è demandata all'indice originario.
- `*amrescan()` richiamata ogniqualvolta c'è un'operazione di confronto su una chiave appartenente all'indice. In particolare, `tde_btree` cifra il valore fornito per prepararlo alla comparazione con il valore cifrato della *Index Key*.

L'unico operatore di comparazione supportato è quello di uguaglianza, in quanto le chiavi sono cifrate con un algoritmo deterministico (AES-256-SIV) che, dato lo stesso input più volte, produce sempre lo stesso valore di output. Non preserva però l'ordinamento, rendendo impossibile l'impiego di *scan* sugli indici con operatori di minoranza e maggioranza.

Limite attributi a lunghezza fissa

All'inizio del progetto il sistema di cifratura degli indici presentava il limite di non poter cifrare le chiavi su colonne con tipo a lunghezza fissa. Questo accadeva perché l'indice *btree* di PostgreSQL riserva per ogni chiave, nelle pagine di memoria degli indici, tanti Byte quanto il tipo della colonna dovrebbe occuparne. Per esempio, se viene creato un indice su una colonna di tipo *int4* verranno riservati esattamente 4 byte per ogni *Index Key* ad essa associata. Tale meccanismo non rendeva possibile la memorizzazione della chiave cifrata il cui peso è quello del tipo originario con l'aggiunta di 16 Byte di *overhead* inseriti dall'algoritmo crittografico.

Questa politica di allocazione della memoria non si applica per le colonne con tipo a lunghezza variabile, proprio per la loro natura di non avere un peso prefissato.

2.3 Gestione delle chiavi crittografiche

Le due chiavi presenti all'interno del contesto di crittografia di `pg_vault_tde` sono gestite da due entità differenti.

2.3.1 DEK

Le chiavi [DEK](#), utilizzate per l'effettiva cifratura delle tuple, sono generate dalla funzione di PostgreSQL `pg_strong_random()`, la quale garantisce la *fork-safety*, ovvero impedisce che due processi di PostgreSQL, che dopo un *fork* condividono la stessa configurazione della libreria OpenSSL, emettano lo stesso numero casuale.

Su tali chiavi è possibile compiere l'operazione di rotazione, ovvero la generazione di una nuova [DEK](#) e la re-cifratura dell'intera tabella associata.

Infine, sono memorizzate nella tabella di catalogo `pg_vault_tde_catalog` creata appositamente, le cui colonne contengono le seguenti informazioni:

- Numero identificativo della tabella;
- Numero di generazione;
- *Wrapped* [DEK](#);
- KMS *provider*;
- Data e ora di creazione;
- Data e ora dell'ultimo aggiornamento;

In particolare, il numero di generazione corrisponde al numero di rotazioni effettuate e viene inizializzato a 1 alla creazione della chiave.

La [DEK](#) memorizzata è detta *wrapped* in quanto viene crittografata dal [KMS provider](#) attraverso la chiave [KEK](#).

Caching delle DEK

Tali chiavi sono memorizzate, in formato crittografico, in *shared memory*,² un'area di memoria fisica simultaneamente accessibile ai vari processi di PostgreSQL, infatti quest'ultimo, a differenza di altri DBMS molto diffusi, adotta un'architettura multi-processo in cui ogni connessione al database viene gestita da un *fork* del processo padre *postmaster*.

All'avvio del server, attraverso l'*hook* `shmem_startup_hook` esposto da PostgreSQL, l'estensione richiede l'allocazione di una parte della *shmem* alla struttura *TdeRelDekCache* così definita:

```
typedef struct TdeRelDekEntry
{
    Oid          relid;
    char         dek[TDE_DEK_LEN];
    char         prev_dek[TDE_DEK_LEN];
    uint64       generation;
    bool         dek_valid;
    bool         prev_dek_valid;
} TdeRelDekEntry;

typedef struct TdeRelDekCache {
    LWLock       lock;
    int          capacity;
    int          used;
    TdeRelDekEntry  entries[FLEXIBLE_ARRAY_MEMBER];
} TdeRelDekCache;
```

Notiamo la presenza del *lock*³ all'interno della struttura per gestire gli accessi concorrenti ad essa.

L'accesso in lettura acquisisce il lock in modalità condivisa `LW_SHARED`, permettendo a tutti i processi *backend* di leggere concorrentemente la chiave; l'accesso in scrittura, acquisisce il lock in modalità esclusiva `LW_EXCLUSIVE`.

L'implementazione di tale struttura permette ai vari processi di accedere direttamente alla **DEK** di una relazione senza dover leggere la tabella `pg_vault_tde_catalog`. In particolare, una chiave crittografica viene caricata all'interno della *cache* al primo *cache miss*, ovvero alla richiesta della **DEK** di una tabella, se questa non è presente nella lista si procede alla lettura della tabella di catalogo e all'inserimento all'interno della *cache*.

Dal punto di vista della sicurezza, le chiavi sono memorizzate in formato *wrapped* in un'area di memoria gestita dal processo PostgreSQL la quale viene automaticamente liberata e pulita in caso di interruzione del server.

²Documentazione PostgreSQL: Shared Memory. URL: <https://www.postgresql.org/docs/current/xfunc-c.html#XFUNC-SHARED-ADDIN>.

³Documentazione PostgreSQL: LWLocks. URL: <https://www.postgresql.org/docs/18/xfunc-c.html#XFUNC-ADDIN-LWLOCKS>.

Il fatto che sia progettata come una lista costringe ad accessi al più $O(n)$ a causa della necessità di scorrere tutto l'*array*.

2.3.2 KEK

La generazione e gestione delle chiavi [KEK](#) può variare in base al [KMS](#) configurato:

- locale, rappresentato da un file in formato PKCS#12 protetto da *password* contenente la chiave. Gli algoritmi di accesso, generazione, *wrapping* e gestione delle chiavi sono implementati da `pg_vault_tde`;
- remoto, compatibile con le [API](#) di HashiCorp Vault e OpenBAO. Non necessita di alcuna implementazione diretta di metodi in quanto tutte le funzioni necessarie sono disponibili tramite chiamate alle [API](#) dei servizi esterni.

Le principali funzionalità che si rendono necessarie per queste chiavi sono:

- *wrap DEK*, che permette di cifrare una chiave [DEK](#) fornita;
- *unwrap DEK*, che permette di decifrare una chiave [DEK](#) cifrata in precedenza;
- *rotate KEK*, che permette di generare una nuova [KEK](#) e re-cifrare tutte le [DEK](#) crittografate in precedenza.

L'astrazione di queste operazioni avviene tramite la *vtable* `TdeKmsProvider`, una struttura in linguaggio C contenente i puntatori a funzione che i singoli *provider* devono obbligatoriamente implementare. Questo pattern architetturale permette all'estensione di supportare backend differenti in maniera trasparente, senza dover modificare il nucleo crittografico. Di seguito è riportata la sua definizione:

```
typedef struct TdeKmsProvider
{
    const char *name;
    bool (*init)(void);
    bool (*wrap_dek)(const unsigned char *dek, int dek_len,
                     unsigned char *wrapped_out, int *out_len);
    bool (*unwrap_dek)(const unsigned char *wrapped, int wrapped_len,
                       unsigned char *dek_out, int *dek_len);
    bool (*rewrap_dek)(const unsigned char *old_wrapped, int old_len,
                       unsigned char *new_wrapped, int *new_len);
    bool (*prepare_kek_rotation)(void);
    void (*commit_kek_rotation)(void);
    bool (*health_check)(void);
    void (*shutdown)(void);
} TdeKmsProvider;
```

All'inizio dello stage, l'estensione prevedeva una singola [KEK](#) per l'intero cluster PostgreSQL.

2.4 Configurazione tramite parametri GUC

La configurazione di `pg_vault_tde` in un'installazione di PostgreSQL ha richiesto la definizione di nuovi parametri GUC. Tali vengono dichiarati come variabili C *extern* in un *header file*, definite successivamente nella funzione `_PG_init()` di ingresso dell'estensione, attraverso il metodo `DefineCustomStringVariable()` della libreria `postgres.h`.

I parametri relativi all'integrazione con HashiCorp Vault⁴ o OpenBao comprendono l'URL dell'endpoint del server, il *namespace* per le installazioni *multi-tenant*, il *token* di autenticazione, il percorso di *mount* del motore *Transit*, il nome della chiave di cifratura, il percorso del certificato CA per la verifica TLS e il *timeout* delle richieste HTTP. Il metodo di autenticazione, configurabile tramite `pg_vault_tde.vault_auth_method`, supporta tre modalità: autenticazione diretta tramite *token*, autenticazione tramite AppRole con *role_id* e *secret_id* e autenticazione tramite le credenziali del *service account* Kubernetes.

Per il provider locale, i parametri configurabili includono il percorso del file *wallet* PKCS#12, che in assenza di configurazione esplicita viene inizializzato ad un valore di default e la passphrase di accesso al *wallet*, la quale non viene mai memorizzata in chiaro in un parametro di configurazione ma in tre modalità mutualmente esclusive, ovvero se è configurata più di una modalità viene segnalato un errore. Queste sono

- `pg_vault_tde.wallet_passphrase_command` comando *shell* il cui output rappresenta la *passphrase*.
- `pg_vault_tde.wallet_passphrase_file` percorso di un file leggibile solo dall'utente proprietario, con permessi 0400 o 0600, che contiene la *passphrase*.
- `pg_vault_tde.wallet_passphrase_env` nome di una variabile d'ambiente che contiene la *passphrase*.

Un'altra variabile GUC degna di nota è il selettore del provider attivo, `pg_vault_tde.kms_provider`, che accetta solamente i valori *vault* e *local* e sulla base del suo valore viene creata la giusta implementazione della *vtable* `TdeKmsProvider`.

Nella Tabella 2.1 è riportato un riepilogo di tutti i parametri GUC introdotti dall'estensione `pg_vault_tde`.

Tabella 2.1: Riepilogo dei parametri GUC di `pg_vault_tde`.

Parametro	Descrizione
<code>pg_vault_tde.vault_url</code>	URL dell'endpoint di HashiCorp Vault o OpenBao.
<code>pg_vault_tde.vault_auth_method</code>	Metodo di autenticazione a Vault (<i>token</i> , <i>approle</i> , <i>kubernetes</i>).
<code>pg_vault_tde.vault_namespace</code>	Namespace per installazioni Vault Enterprise multi-tenant.
<code>pg_vault_tde.vault_token</code>	Token di autenticazione per Vault.
<code>pg_vault_tde.vault_transit_mount</code>	Percorso di mount dell'engine Transit in Vault.
<code>pg_vault_tde.vault_key_name</code>	Nome della chiave KEK su Vault.

⁴ Documentazione HashiCorp Vault. URL: <https://developer.hashicorp.com/vault/docs>.

Tabella 2.1

Parametro	Descrizione
<code>pg_vault_tde.vault_ca_cert</code>	Percorso del certificato CA per la verifica TLS di Vault.
<code>pg_vault_tde.vault_timeout_ms</code>	Timeout in millisecondi per le chiamate HTTP a Vault.
<code>pg_vault_tde.vault_role_id</code>	Role ID per l'autenticazione AppRole.
<code>pg_vault_tde.vault_secret_id</code>	Secret ID per l'autenticazione AppRole.
<code>pg_vault_tde.vault_role_name</code>	Nome del ruolo AppRole per la rotazione automatica del Secret ID.
<code>pg_vault_tde.vault_k8s_role</code>	Nome del ruolo di autenticazione Kubernetes in Vault.
<code>pg_vault_tde.vault_k8s_mount</code>	Percorso di montaggio dell'engine di autenticazione Kubernetes.
<code>pg_vault_tde.bgw_enabled</code>	Abilita il background worker per il rinnovo automatico del token.
<code>pg_vault_tde.enabled</code>	Abilita o disabilita globalmente la cifratura dell'estensione.
<code>pg_vault_tde.dek_cache_ttl</code>	TTL in secondi della cache locale delle DEK.
<code>pg_vault_tde.crypto_provider</code>	Provider OpenSSL 3.x per l'accelerazione crittografica hardware.
<code>pg_vault_tde.token_renewal_interval</code>	Intervallo in secondi tra i tentativi di rinnovo del token.
<code>pg_vault_tde.kms_provider</code>	Provider KMS attivo: <code>vault</code> o <code>local</code> .
<code>pg_vault_tde.wallet_path</code>	Percorso del file wallet PKCS#12.
<code>pg_vault_tde.wallet_auto_open</code>	Apri automaticamente il wallet all'avvio se la passphrase è disponibile.
<code>pg_vault_tde.max_encrypted_relations</code>	Numero massimo di DEK di relazioni tenute in cache.
<code>pg_vault_tde.toast_encryption</code>	Abilita la cifratura delle tabelle TOAST associate a tabelle cifrate.
<code>pg_vault_tde.wallet_passphrase_env</code>	Variabile d'ambiente contenente la passphrase del wallet.
<code>pg_vault_tde.wallet_passphrase_file</code>	File contenente la passphrase del wallet (permessi 0400/0600).
<code>pg_vault_tde.wallet_passphrase_command</code>	Comando shell il cui output fornisce la passphrase del wallet.
<code>pg_vault_tde.wallet_dev_mode_passphrase</code>	Passphrase in chiaro per ambienti di sviluppo/CI.
<code>pg_vault_tde.dev_mode</code>	Abilita la modalità sviluppo/CI con passphrase in chiaro.

Capitolo 3

Sviluppo

3.1 Crittografia delle tabelle

La prima attività di sviluppo svolta ha riguardato il limite citato in precedenza di non crittazione delle tabelle [TOAST](#). Una soluzione era necessaria in quanto demandare a *heap* l’inserimento di questi attributi rendeva illeggibile la tabella e provocava, a causa dei dati memorizzati in chiaro sul disco, una falla di sicurezza importante.

Un’ipotesi di soluzione si è sviluppata nell’invocazione del metodo `heap_toast_insert_or_update()` con la tupla criptata come parametro. Fin da subito questo approccio si è rivelato non risolutivo in quanto applicare il [TOAST](#) dopo la cifratura porta alla perdita di efficienza della compressione, la quale si basa sulla rimozione di sequenze di caratteri ripetuti. La cifratura per definizione è ad alta entropia, quindi le ripetizioni sono rare.

3.1.1 Flusso di inserimento *TOAST*

La soluzione definitiva è partita dall’analisi del flusso che *heap* adotta per l’inserimento delle tuple con attributi che potrebbero richiedere *toasting*, visibile nella [Figura 3.1](#), e da una sua reimplementazione parziale da parte del [TAM](#) dell’estensione. Questo si è reso necessario in quanto, come riportato in precedenza, *heap* non permette di intervenire all’interno della sua esecuzione. L’obiettivo era riuscire a cifrare l’intera tupla dopo la compressione e, se questa era in grado di essere memorizzata nella stessa pagina della tabella padre (quindi con lunghezza pari o inferiore a 2 KB), salvarla *in-line* altrimenti *out-of-line* ovvero cifrare ogni *chunk* dopo la suddivisione della tupla in questi.

Cifrare prima della divisione dei dati avrebbe determinato la modifica della dimensione del *Datum*, l’astrazione di qualsiasi tipo di dato presente in PostgreSQL, in quanto il processo crittografico aggiunge un *overhead*, e di conseguenza l’invalidazione del calcolo matematico per la frammentazione e l’impossibilità di leggere correttamente la tupla.

I metodi da sostituire per garantire il corretto funzionamento sia dell’inserimento sia del successivo recupero degli attributi erano:

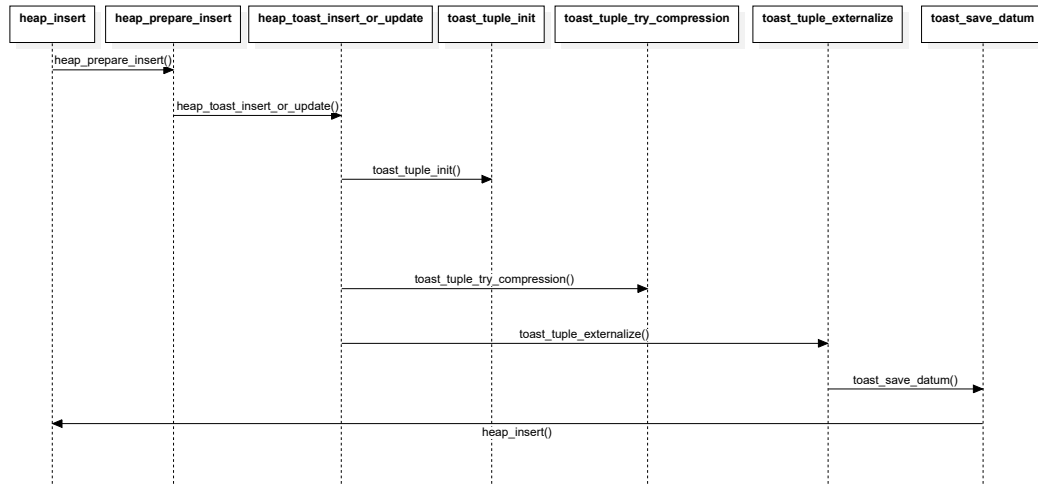


Figura 3.1: Flusso di *heap* all'inserimento di una tupla che necessita di toast

- `heap_toast_insert_or_update()`, punto di ingresso e di controllo dell'inserimento *toast*;
- `toast_tuple_externalize()`, metodo dedicato al *dispatch* verso la funzione di *chunking*;
- `toast_save_datum()`, funzione deputata esclusivamente alla suddivisione del valore in *chunk* e alla loro scrittura nella relazione fisica di *TOAST* (*out-of-line*), restituendo il puntatore esterno di tipo `varatt_external` da inserire nella tupla padre.

3.1.2 Soluzione definitiva

Invece di fare affidamento sull'intercettazione dei chunk tramite il dispatch del TAM (scalato nativamente dal motore TOAST di PostgreSQL in scrittura), si è proceduto a effettuare un vero e proprio *fork* del percorso di scrittura e compressione dei dati TOAST. Una visualizzazione del percorso di scrittura è presente nella Figura 3.2.

L'estensione interviene direttamente a livello di TAM durante le operazioni di scrittura (`tuple_insert`, `tuple_update`, ecc.). Il controllo viene deviato alla funzione personalizzata `pg_vault_tde_insert_or_update`, la quale replica il funzionamento dell'analogo metodo di *heap*, rilevando se la tupla supera la soglia di memorizzazione in pagina (`TOAST_TUPLE_THRESHOLD = 2 KB`).

Successivamente, all'interno della funzione `pg_vault_tde_toast_tuple` vengono rilevati tutti gli attributi eleggibili all'applicazione del TOAST. Per ognuna di queste colonne, viene applicata una strategia diversa in base al loro tipo di *storage*¹:

- Se hanno *storage* di tipo `EXTENDED`, viene tentata la compressione con il metodo *core* `toast_tuple_try_compression` e, nel caso in cui la lunghezza sia ancora maggiore di

¹Documentazione PostgreSQL TOAST. URL: <https://www.postgresql.org/docs/current/storage-toast.html>.

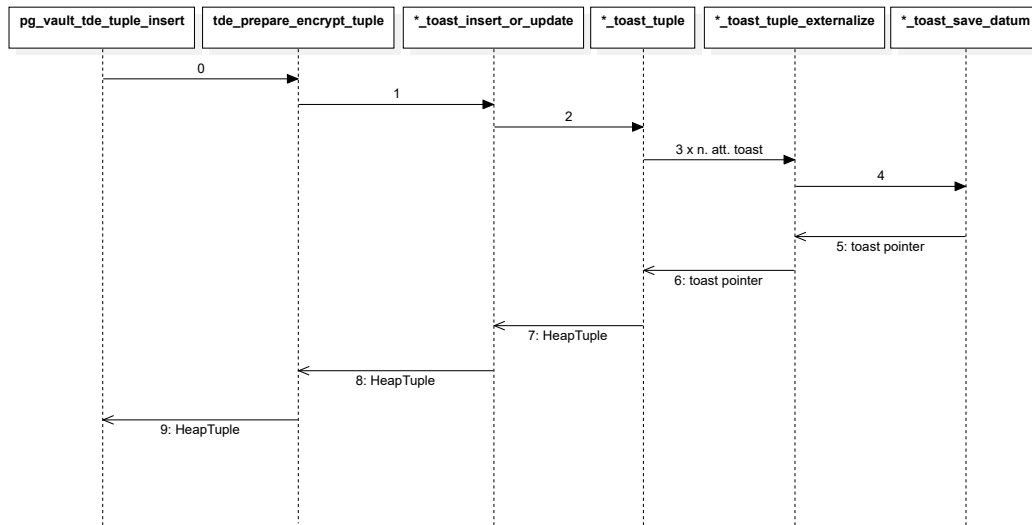


Figura 3.2: Flusso di *encrypted_heap* all'inserimento di una tupla che necessita di toast.

(Nota: * sostituisce il prefisso `pg_vault_tde`)

`TOAST_TUPLE_THRESHOLD`, il dato viene memorizzato *out-of-line* attraverso l'invocazione della funzione `pg_vault_tde_toast_tuple_externalize`, la quale demanda a sua volta il *chunking* alla funzione `pg_vault_tde_toast_save_datum`;

- Se hanno *storage* di tipo `EXTERNAL`, la compressione viene saltata, forzando la memorizzazione *out-of-line*;
- Se hanno *storage* di tipo `MAIN`, viene effettuata la compressione e tentata la memorizzazione in riga. Se questo non è possibile si ricorre all'*out-of-line*;
- Se hanno *storage* di tipo `PLAIN`, i dati devono essere memorizzati nella stessa pagina della tupla, escludendo sia la compressione che lo spostamento fuori riga. Nel caso in cui questo non sia possibile, viene segnalato un errore.

La compressione avviene per mezzo dell'algoritmo PGLZ o LZ4, è PostgreSQL stesso a decidere quale utilizzare. È però importante notare che la compressione viene eseguita prima della crittazione e memorizzazione del dato, sia che avvenga in pagina che fuori, in modo da mantenere il vantaggio dell'applicazione di questi algoritmi senza però perdere la sicurezza fornita dalla crittografia. Come citato in precedenza, comprimere dopo la cifratura impedirebbe di rimuovere le ridondanze dai dati risultando in una pesantezza generale del database molto maggiore.

Gli attributi destinati alla memorizzazione esterna vengono divisi in *chunk* di dimensione predefinita (`TOAST_MAX_CHUNK_SIZE`) dalla funzione `pg_vault_tde_toast_save_datum`, la quale provvede alla costruzione della tupla contenente il frammento e al suo inserimento, attraverso `heap_insert`, preceduto dall'invocazione del metodo `tde_encrypt_heap_tuple`. La tabella che ospita tali frammenti è la cosiddetta *toast table*, generata contestualmente

alla creazione della tabella cifrata. Ogni tupla contenuta in questa particolare relazione è composta dai seguenti attributi:

- *Chunk Id* numero identificativo assegnato all'intero dato frammentato;
- *Chunk Seq* numero di sequenza del singolo frammento;
- *Chunk Data* contenuto effettivo del *chunk*.

La coppia *id* e numero di sequenza forma la *primary key* della tabella.

AM delle tabelle *toast*

Grazie all'assegnazione dell'access method `encrypted_heap` anche alle relazioni TOAST, il percorso di lettura nativo di PostgreSQL (`heap_fetch_toast_slice`) esegue automaticamente la decifratura dei singoli chunk. Questo accade in quanto i metodi del *core* chiamano le *callback* della struttura *TableAmRoutine* configurata e non i metodi "concreti" di *heap*.

Chiavi DEK dei *chunk*

Per mantenere una tabella di catalogo snella si è scelto di cifrare i singoli *chunk* con l'algoritmo AES-256-GCM utilizzando la chiave della tabella genitore. Il recupero di questa avviene all'interno della funzione `pg_vault_tde_kms_get_rel_dek` attraverso l'estrazione dell'attributo proprio di ogni tabella, `relkind`, che identifica il tipo di tabella passata come parametro al metodo. Nel caso in cui quest'ultima sia una *toast table*, avviene un'invocazione ricorsiva sulla tabella padre.

3.1.3 CLUSTER e VACUUM FULL

Dopo aver terminato l'implementazione della soluzione al problema degli attributi *toast* è sorto un problema non considerato in precedenza.

Durante l'esecuzione di operazioni di manutenzione come `VACUUM FULL`² e `CLUSTER`, PostgreSQL necessita di riscrivere l'intera tabella all'interno di un nuovo file fisico, riorganizzando lo spazio ed eliminando le tuple non più visibili. Nell'architettura di PostgreSQL, questo compito è affidato alla *callback* della *TableAmRoutine* `relation_copy_for_cluster`.

Tuttavia, l'implementazione standard di *heap*, utilizzata fino a quel momento, assume che i dati risiedano in chiaro. Eseguita su tabelle cifrate, questa funzione provocava un arresto anomalo del server. Tale arresto era causato da due motivi distinti:

- La lettura delle tuple della tabella da riscrivere passava attraverso la *callback* personalizzata dell'estensione `table_beginscan`, la quale estrae e decifra la tupla in memoria ma rilascia il *pin* sul buffer fisico della pagina. Quest'ultimo si riferisce alla tecnica di lettura, utilizzata da gran parte dei DBMS, in cui i blocchi di dati sono caricati all'interno del *Buffer Pool*, un'area riservata in RAM, e fissati con un *pin* per segnalare

²Documentazione PostgreSQL `VACUUM`. URL: <https://www.postgresql.org/docs/current/sql-vacuum.html>.

al *Buffer Manager* di PostgreSQL di mantenerli in memoria fino al termine della transazione.

`pg_vault_tde` non può tenere il *pin* su tale *buffer* perché risulterebbe in uno spreco di memoria in quanto la decifratura avviene in un'area di memoria che per motivi di sicurezza è controllata dall'estensione e non nel *Buffer Pool*. Infatti se la decifratura avvenisse all'interno di quest'ultimo spazio, il *Buffer Manager* marcherebbe lo *slot* come modificato (*dirty*) e al momento del rilascio del *pin*, verrebbe riscritto il suo contenuto sulle pagine del disco fisso. Si andrebbe incontro al rischio di violazione della cifratura.

Successivamente alla lettura e al rilascio del *pin*, la funzione

`HeapTupleSatisfiesVacuum`, il cui compito è determinare se una tupla può essere definitivamente eliminata dal processo di riscrittura, riceve un riferimento a un *buffer* non valido e causa un *crash* volontario definito in essa, denominato *panic*.

- Quando viene eseguito il comando `VACUUM` o `CLUSTER`, il sistema si aspetta di trovare tutte le tuple e i relativi attributi sulla stessa riga logica, caricata in memoria (*HeapTuple*). Proprio per questo, la funzione di *heap* invocava il metodo `toast_flatten_tuple` sul payload cifrato, che interpretava dei byte casuali all'interno di esso come puntatore al *toast*, risultando in un *crash* per *wild-pointer dereference*.

Per risolvere queste problematiche, si è reso necessario scrivere una nuova funzione con cui sovrascrivere la *callback* del TAM: `pg_vault_tde_relation_copy_for_cluster`. Questa, invece di utilizzare `table_beginscan`, avvia una scansione fisica tramite `heap_beginscan` e `heap_getnext`, metodi di *heap*, i quali impediscono la decifratura immediata operata da `table_scan` e mantengono valido il *pin* sul buffer, consentendo a `HeapTupleSatisfiesVacuum` di verificare lo stato della tupla ancora cifrata in sicurezza. Una volta terminata la verifica, se la riga risulta ancora visibile, viene decifrata, i suoi attributi esterni, se presenti, sono portati *in-line* e infine la tupla viene ricifrata con la sua chiave già esistente e riscritta mediante la funzione `rewrite_heap_tuple`.

3.2 Utility di backup logico

L'architettura di crittografia TDE dei dati sviluppata per PostgreSQL opera in modo, appunto, trasparente alle applicazioni client, intervenendo a livello di memorizzazione fisica. Quando un utente esegue un'interrogazione SQL come una `SELECT`, l'*Executor* chiama automaticamente le *routine* definite dalla *TableAmRoutine* della tabella in questione, che decifrano i dati in memoria e li restituiscono all'utente in chiaro. L'*utility* nativa per l'esportazione logica di PostgreSQL, `pg_dump`,³ estrae lo schema e i dati del database effettuando normali query; per questo motivo, il backup logico viene salvato sul disco interamente in chiaro.

³Documentazione PostgreSQL `pg_dump`. URL: <https://www.postgresql.org/docs/current/app-pgdump.html>.

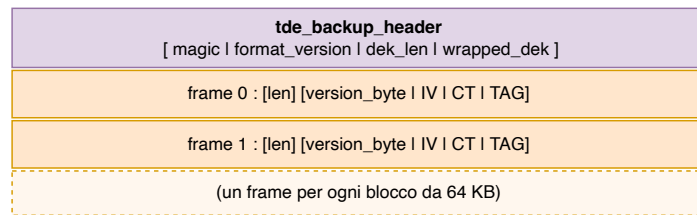


Figura 3.3: Formato del file di backup cifrato

Questo può rappresentare un potenziale rischio di sicurezza, ma va considerato il fatto che se un soggetto è in grado di utilizzare `pg_dump` per estrarre i dati da un *database*, allora sarà in grado anche di accedervi e recuperare i dati direttamente da esso.

Per fornire agli utenti un'alternativa cifrata a `pg_dump` per poter memorizzare i backup logici anch'essi cifrati, è stato progettato e implementato un sistema composto da due strumenti denominati `pg_dump_tde` e `pg_restore_tde`, che si integrano con le rispettive *utility* native di PostgreSQL.

3.2.1 Architettura

Invece di duplicare la complessa logica interna di esportazione e importazione di PostgreSQL, queste *utility* comunicano attraverso una *pipe* con `pg_dump` e `pg_restore`, cifrando e decifrando il flusso di dati in modalità streaming.

Formato di backup

Il formato del file risultante, visibile in [Figura 3.3](#), è composto da un *header*, unico per ogni backup, che contiene:

- *Magic* rappresenta una firma del tipo di file. Per ogni backup è uguale alla stringa di 10 caratteri `PGVAULTTDE`. È necessaria durante il *restore* per scartare immediatamente file non idonei;
- *Format version* serve per versionare il formato, in previsione di future modifiche e identificare i backup con formato precedente;
- *Wrapped dek* e *Dek length* sono la chiave utilizzata per la cifratura del backup, a sua volta cifrata, e la sua lunghezza.

Inoltre, ogni esportazione logica è cifrata a blocchi, cioè prima dell'applicazione dell'algoritmo di criptazione, l'output di `pg_dump` viene diviso in blocchi di 64 KB poi cifrati singolarmente.

Flusso di esecuzione: *dump*

Quando viene lanciato il comando `pg_dump_tde`, l'*utility* avvia una connessione con il server PostgreSQL tramite la libreria `libpq`. Questa connessione è finalizzata esclusivamente al

recupero delle variabili di configurazione del [KMS](#) attivo nel database, tramite l'esecuzione delle query `SHOW <nome_variabile_GUC>`. Dopo aver ottenuto queste informazioni viene costruita una rappresentazione del provider, interna a `pg_dump_tde`, speculare a quella interna al database. Viene poi richiesto a quest'ultimo di generare una chiave [DEK](#) temporanea ed esclusiva per il backup corrente, che a sua volta, come accade nel DBMS, viene cifrata con la chiave [KEK](#) del [KMS](#). Il testo cifrato risultante viene scritto nell'*header* del file di esportazione. L'intestazione, denominata `tde_backup_header`, viene scritta all'inizio del file di destinazione con una dimensione fissa di 532 Byte. Successivamente, l'*utility* esegue una chiamata di sistema `fork` per creare un processo figlio. Il processo figlio devia lo standard output su una *pipe* anonima ed esegue il binario nativo `pg_dump` forzando l'uso del formato personalizzato (`-Fc`), l'unico che produce una sequenza di byte adatta alla cifratura in streaming. Il processo padre legge in parallelo i dati in chiaro provenienti dalla *pipe* in blocchi di dimensione fissa pari a 64 KB, cifrando ciascun blocco in modalità **AES-256-GCM** per poi scriverlo nel file di destinazione preceduto da un intero a 32 bit che ne indica la dimensione complessiva comprensiva dell'*overhead* crittografico.

Flusso di esecuzione: *restore*

L'*utility* `pg_restore_tde` legge per prima cosa l'intestazione `tde_backup_header` dal file cifrato, ne valida la versione ed estrae la chiave *wrapped dek*. Costruisce la stessa astrazione del [KMS](#) di `pg_dump_tde` e contatta quindi il provider [KMS](#) attivo per decifrare la DEK. Al termine di questo, avvia un processo figlio tramite `fork` che esegue il binario nativo `pg_restore`, collegandone lo standard input alla *pipe* di decifratura. Il processo padre legge sequenzialmente i blocchi cifrati dal file di input, ne convalida l'integrità e li decifra prima di immetterli nella *pipe*, permettendo al ripristino di completarsi in modo del tutto trasparente per il server di destinazione.

3.2.2 Sicurezza del backup

Un aspetto critico affrontato in fase di progettazione riguarda la vulnerabilità ad attacchi di manipolazione del flusso, come il riordinamento o il troncamento dei blocchi cifrati da parte di un soggetto esterno. Per neutralizzare questa minaccia, si è sfruttata la capacità di AES-GCM di autenticare dati aggiuntivi non cifrati, i cosiddetti [Additional Authenticated Data \(AAD\)](#). Durante la cifratura di ogni blocco, il numero di sequenza dello stesso viene codificato come intero a 64 bit in formato *big-endian* tramite la funzione `pg_hton64` e passato come parametro [AAD](#) alla libreria crittografica OpenSSL. Di conseguenza, il tag di autenticazione finale del blocco valida non solo il contenuto cifrato ma anche la sua posizione nel flusso. Qualsiasi tentativo di alterare l'ordine dei blocchi o di interrompere anticipatamente la trasmissione provocherà un fallimento immediato dell'autenticazione durante il ripristino, bloccando l'operazione prima che tuple parziali o fuori sequenza possano essere scritte nel database.

Per quanto riguarda la [Inter-Process Communication \(IPC\)](#), la *pipe* creata tra le *utility* native e quelle custom è anonima, ovvero non ha alcun nome riconoscibile sul *filesystem*, quindi non è leggibile da utenti non privilegiati. Il fatto che sia intercettabile dall'utente *root*, attraverso per esempio un *ptrace*, rappresenta un rischio che però non è parte del *threat model* della [TDE](#).

3.2.3 Test e Validazione

La stabilità del sistema di backup logico è validata da una suite di test automatici basata sul framework [TAP](#) di PostgreSQL. I test ricreano un ambiente di test completo inizializzando un'istanza PostgreSQL cifrata e simulando un server HashiCorp Vault. Si verifica che il file generato da `pg_dump_tde` risulti effettivamente cifrato ed esente da stringhe in chiaro riconducibili ai dati originali. Lo script di test modifica deliberatamente il file di dump cifrato in vari modi prima del ripristino:

- Scambiando di posizione due blocchi all'interno del file.
- Modificando singoli byte del payload cifrato.
- Troncando il file.

In ciascuno di questi casi, viene verificato che l'*utility* `pg_restore_tde` fallisca, interrompendo l'operazione ed evitando la scrittura di dati parziali o fuori sequenza. Infine viene eseguito un ripristino completo tramite `pg_restore_tde` su un database vuoto e verificata la presenza delle tabelle e dei dati in esse contenuti, in precedenza esportati.

3.3 Crittografia degli indici

La cifratura degli indici rappresenta una delle sfide tecniche più significative nello sviluppo di `pg_vault_tde`. In un database relazionale, gli indici B-Tree richiedono un ordinamento totale delle chiavi per poter eseguire ricerche binarie e mantenere l'albero bilanciato. L'adozione di un algoritmo di cifratura probabilistico come AES-256-GCM utilizzato a livello di tabella dal [TAM](#), non è applicabile agli indici: a causa dell'uso di vettori di inizializzazione ([IV](#)) casuali, valori in chiaro identici produrrebbero testi cifrati differenti, rendendo impossibile qualsiasi operazione di confronto e ricerca sull'indice.

Per superare questa limitazione, `pg_vault_tde` implementa uno [IAM](#) personalizzato denominato `tde_btree`, il quale crea un nuovo indice, basato su *btree* di PostgreSQL, che introduce una cifratura deterministica e autenticata basata sull'algoritmo **AES-256-SIV** (Synthetic [IV](#)).⁴

⁴Dan Harkins. *RFC 5297: Synthetic Initialization Vector (SIV) Block Cipher Mode for Key Wrapping and Authenticated Encryption*. 2008. URL: <https://datatracker.ietf.org/doc/html/rfc5297>.

3.3.1 *Operator class e operator family*

Come citato in precedenza, la versione iniziale di `pg_vault_tde` limitava la cifratura delle chiavi dell'indice `tde_btree` esclusivamente ai tipi di dato a lunghezza variabile, come *text* o *bytea*. Questo era dettato dal modo in cui PostgreSQL alloca lo spazio nella pagina per le *index key*. Per i tipi a lunghezza fissa riserva rigidamente lo spazio corrispondente, ad esempio per una chiave con tipo *int4* riserva esattamente 4 Byte. Al contrario, per i tipi a lunghezza variabile, prima di allocare la memoria, legge la lunghezza del valore.

The limite è stato risolto grazie all'introduzione di una *operator family*⁵ che racchiude tutte le *operator class* e funzioni di supporto (*comparison*) operanti sulle chiavi dell'indice custom.

In particolare, una *operator class* definisce le operazioni e il modo con il quale indicizzare una colonna di uno specifico tipo di dato. Essa permette di specificare:

- *index type* a cui si riferisce;
- *operator family* di appartenenza;
- *operators*, ovvero gli operatori SQL che l'indice è in grado di ottimizzare;
- *storage*, il tipo di dato effettivo memorizzato sul disco.

Un esempio di *operator class* creata nel contesto di `tde_btree` è la seguente:

```
CREATE OPERATOR CLASS tde_int4_enc_ops
DEFAULT FOR TYPE int4
USING tde_btree
FAMILY tde_enc_ops_family
AS
OPERATOR 3 = (int4, int4),
STORAGE bytea;
```

Le *operator family* invece, sono dei raggruppamenti di *operator class*. Nascono dalla necessità di estendere il confronto tra tipi differenti. Permettono di definire le *support functions* ovvero le funzioni C che l'indice può utilizzare in fase di ricerca o in fase di inserimento di una tupla. Ad esempio, per *btree*, possiamo specificare la funzione di confronto tra due nodi.

L'*operator family* creata per `tde_btree` è la seguente:

```
CREATE OPERATOR FAMILY tde_enc_ops_family USING tde_btree;

ALTER OPERATOR FAMILY tde_enc_ops_family USING tde_btree
ADD FUNCTION 1 (int4, int4) tde_int4_enc_cmp(int4, int4);
```

⁵ Documentazione PostgreSQL: Interfacing Extensions to Indexes. URL: <https://www.postgresql.org/docs/current/xindex.html>.

```

ALTER OPERATOR FAMILY tde_enc_ops_family USING tde_btree
    ADD FUNCTION 1 (int8, int8) tde_int8_enc_cmp(int8, int8);

ALTER OPERATOR FAMILY tde_enc_ops_family USING tde_btree
    ADD FUNCTION 1 (uuid, uuid) tde_uuid_enc_cmp(uuid, uuid);

ALTER OPERATOR FAMILY tde_enc_ops_family USING tde_btree
    ADD FUNCTION 1 (date, date) tde_date_enc_cmp(date, date);

ALTER OPERATOR FAMILY tde_enc_ops_family USING tde_btree
    ADD FUNCTION 1 (timestampz, timestampz)
        tde_timestampz_enc_cmp(timestampz, timestampz);

```

Possiamo notare come per ogni tipo di dato a lunghezza fissa supportato sia presente la dicitura `FUNCTION 1`, la quale indica la funzione fondamentale di supporto dell'indice. Nel caso dell'indice B-Tree, essa rappresenta una funzione di comparazione, mentre nel caso dell'indice Hash⁶ si tratta della funzione che calcola l'hash del valore in ingresso. Nel caso di `tde_btree`, in modo analogo a B-Tree, essa indica la funzione di confronto per il tipo di dato corrispondente. Per ogni operatore è così definita:

```

CREATE FUNCTION tde_int4_enc_cmp(int4, int4)
RETURNS integer LANGUAGE C STRICT IMMUTABLE
AS 'MODULE_PATHNAME', 'tde_enc_bytea_cmp';

```

Notiamo che la funzione C invocata è sempre la stessa `tde_enc_bytea_cmp`, un metodo di confronto tra stringhe di byte.

Nonostante le funzioni di confronto per ogni tipo di dato siano già definite da *btree*, si è reso necessario creare queste nuove funzioni che invocano tutte lo stesso metodo sottostante, in quanto i metodi già implementati confrontano il valore, anche criptato, interpretato come tipo di dato e non come sequenza di Byte.

3.3.2 AES-256-SIV

L'algoritmo AES-SIV è deterministico, ovvero a parità di chiave di cifratura e testo in chiaro, produce sempre lo stesso testo cifrato. Questo consente di confrontare testi cifrati per uguaglianza, rendendo possibili le ricerche con tali operatori (`=`, `IN`, clausole `ON CONFLICT`). Rispetto ad altri approcci deterministici come AES-ECB, AES-SIV garantisce proprietà di autenticazione e sicurezza notevolmente superiori (è resistente all'abuso dei vettori di inizializzazione, *misuse-resistant*) e impedisce attacchi basati sulla ripetizione delle chiavi.

È importante notare come AES-256-SIV necessiti di una chiave di lunghezza 512 bit (64 Byte), che viene divisa in due parti di 256 bit ciascuna. La prima viene utilizzata per il

⁶Documentazione PostgreSQL: Hash Indexes. URL: <https://www.postgresql.org/docs/current/indexes-types.html#INDEXES-TYPES-HASH>.

calcolo del SIV mentre la seconda per l'effettiva cifratura del dato. Si rende quindi necessario fornire una chiave di cifratura **DEK** dedicata a ogni indice per la cifratura delle *Index Key* e un metodo per espandere questa chiave di 32 Byte in 64 Byte e garantire il determinismo di tale operazione, altrimenti ad ogni derivazione si otterrebbe un **IV** diverso, perdendo il vantaggio di AES-256-SIV. Questo viene reso possibile dalla funzione di derivazione PKCS5_PBKDF2_HMAC con *salt* fisso e una singola iterazione:

```
PKCS5_PBKDF2_HMAC(dek, TDE_DEK_LEN,
                  (unsigned char *)"tde-siv", 7,
                  1,
                  EVP_sha256(), 64, siv_key);
```

3.3.3 Architettura di tde_btree

Anche qui, invece di riscrivere da zero la complessa logica di gestione dei B-Tree, l'estensione incapsula il metodo nativo **btree** di PostgreSQL. Durante l'inizializzazione dell'estensione (**tde_iam_init()**), la struttura di callback **IndexAmRoutine** dell'indice di PostgreSQL viene clonata all'interno di una struttura **tde_btree_methods**, sovrascrivendo cinque funzioni critiche per la manipolazione delle chiavi:

- ***ambuild()** invocata alla costruzione dell'indice, delega completamente la costruzione alla funzione nativa di *btree*, impostando tuttavia un flag globale **tde_iam_build_in_progress = true**. In questo modo durante la scansione sequenziale della tabella per la costruzione dell'indice, la *callback* del **TAM** **pg_vault_tde_index_build_range_scan**, richiamata dai metodi di *btree*, legge il flag attivo e provvede a cifrare le chiavi di indicizzazione prima che vengano inserite nel buffer di ordinamento del B-Tree.
- ***aminsert()** invocata all'inserimento di nuove tuple con colonne indicizzate. Cifra i valori delle chiavi e delega l'effettivo inserimento nella pagina fisica dell'indice alle routine standard del B-Tree.
- ***amrescan()** invocata quando viene eseguita una ricerca mediante l'indice. Legge i valori utilizzati dalla scansione e se l'operatore di ricerca corrisponde a un'uguaglianza allora il valore della chiave di ricerca viene cifrato da AES-SIV con la **DEK** della tabella indice (*Index Relation*). Successivamente, la funzione di confronto interna viene invocata consentendo di verificare la corrispondenza esatta dei byte della chiave di ricerca e della *Index Key*.
- ***amvalidate()** il cui compito è quello di validare la correttezza logica e strutturale di una *operator family*.

Le funzioni di scansione **ambeginscan** e **amendscan** non necessitano di elaborazioni crittografiche e vengono delegate direttamente alle implementazioni native.

3.3.4 Limitazioni

La crittografia degli indici in `pg_vault_tde` comporta alcune limitazioni che devono essere considerate. Le scansioni ad intervallo (*range scan*), non sono possibili, in quanto l'algoritmo AES-256-SIV non preserva l'ordinamento originale. Di conseguenza, le query che utilizzano operatori di disuguaglianza o intervallo (`>`, `<`, `BETWEEN`) forzano PostgreSQL a ricorrere alla scansione sequenziale, con decifratura di ogni tupla, dell'intera tabella.

Inoltre, le *index-only scans*, ovvero le scansioni che restituiscono i dati richiesti dalla query direttamente dalle pagine degli indici e dalle *index key* senza accedere alla tabella, sono state disattivate impostando a `NULL` il puntatore a funzione `amcanreturn` nella struttura `IndexAmRoutine`. Se questa modalità fosse abilitata, verrebbe restituito al client il valore cifrato delle chiavi di indice, in quanto la decifratura avviene esclusivamente nel layer `TAM`, durante l'accesso a una tabella.

3.4 Auditing

L'*auditing* rappresenta un requisito fondamentale per qualsiasi sistema che gestisca dati sensibili.⁷ In particolare con tale termine si intende la tracciatura degli eventi crittografici come accessi, rotazioni e modifiche. Senza questa sarebbe impossibile risalire al responsabile di particolari azioni e quindi rispettare il principio di *accountability*.⁸

Per questo motivo, si è reso necessario integrare in `pg_vault_tde` un sistema di auditing costantemente attivo e che non è possibile disabilitare.

L'approccio utilizzato per tale sistema è quello degli *hook*. In un *header file* è definito

```
typedef void(*tde_audit_hook)(TdeAuditEvent event,
                             const char*   reloid,
                             bool          success);

extern tde_audit_hook audit_hook_ptr;

static inline void
tde_audit(TdeAuditEvent event, const char *reloid, bool success)
{
    if (audit_hook_ptr)
        (*audit_hook_ptr)(event, reloid, success);
}
```

⁷Elaine Barker. *Recommendation for Key Management: Part 1 – General*. NIST Special Publication 800-57 Part 1 Revision 5. Gaithersburg, MD, USA: National Institute of Standards e Technology, mag. 2020. URL: <https://doi.org/10.6028/NIST.SP.800-57pt1r5>.

⁸Regolamento (UE) 2016/679 del Parlamento europeo e del Consiglio, del 27 aprile 2016, relativo alla protezione delle persone fisiche con riguardo al trattamento dei dati personali, nonché alla libera circolazione di tali dati (Regolamento generale sulla protezione dei dati). 2016. URL: <https://eur-lex.europa.eu/legal-content/IT/TXT/?uri=CELEX:32016R0679>.

Il puntatore definito *extern* viene poi inizializzato in fase di avvio dell'estensione come

```
audit_hook_ptr = tde_audit_handler;

static void tde_audit_handler(TdeAuditEvent event, const char* reloid,
                             bool success)
{
    const char *rolname = OidIsValid(GetUserId())
        ? GetUserNameFromId(GetUserId(), true)
        : "(system)";

    ereport(LOG,
            (errmsg("AUDIT: event=%s, oid=%s, user=%s, success=%s, pid=%d",
                  tde_event_string(event),
                  reloid != NULL ? reloid : "-",
                  rolname != NULL ? rolname : "(unknown)",
                  success ? "t" : "f",
                  MyProcPid),
              errhidestmt(true),
              errhidecontext(true)
            )
    );
}
```

Successivamente, alla conclusione di ogni operazione che richiede auditing viene richiamato l'hook `tde_audit(Event, RelOid, Success)`.

3.4.1 Formato dei messaggi

Ogni evento viene scritto nel log del server PostgreSQL con severità `LOG` tramite la funzione `ereport(LOG)`. Il messaggio segue un formato fisso e leggibile da strumenti automatici:

```
AUDIT: event=<nome>, oid=<rel_oid>, user=<utente>, success=<t|f>, pid=<pid>
```

Il campo `event` identifica l'operazione eseguita, il campo `oid` riporta l'identificativo della relazione coinvolta, oppure il simbolo `-` per gli eventi a livello di cluster. Il campo `success` vale `t` in caso di successo e `f` in caso di fallimento. Il campo `pid` identifica il processo PostgreSQL che ha generato l'evento.

3.4.2 Eventi registrati

Il sottosistema copre l'intero ciclo di vita delle chiavi crittografiche e delle operazioni DDL sulle relazioni cifrate. Un riepilogo è presente in [Tabella 3.1](#).

Tabella 3.1: Eventi di *audit* registrati da `pg_vault_tde`

Evento	Descrizione
AUDIT_LOG_START	Sottosistema di audit inizializzato all'avvio del server.
AUDIT_LOG_STOP	Sottosistema di audit arrestato allo spegnimento.
DEK_ACCESS	DEK letta dalla cache in memoria condivisa o dal KMS.
DEK_CREATE	Nuova DEK per-relazione registrata.
DEK_UPDATE	DEK aggiornata per-relazione.
DEK_ROTATE	DEK per-relazione ruotata.
DEK_DELETE	DEK rimossa dal catalogo.
KEK_ROTATE	KEK ruotata.
KMS_AUTH_SUCCESS	Autenticazione verso il KMS riuscita.
KMS_AUTH_FAILURE	Autenticazione verso il KMS fallita.
WALLET_OPEN	Wallet locale aperto.
WALLET_CLOSE	Wallet locale chiuso, DEK rimosse dalla cache.
RELATION_ENCRYPT	Relazione convertita in <code>encrypted_heap</code> .
RELATION_DECRYPT	Relazione riconvertita in heap ordinario.
ACCESS_DENIED	Decifratura fallita per chiave errata o permesso mancante.
INTEGRITY_VIOLATION	Fallimento del tag GCM o del checksum.

3.5 Attività di manutenzione

L'evoluzione di `pg_vault_tde` ha richiesto, parallelamente allo sviluppo delle nuove funzionalità, un insieme di attività di manutenzione e miglioramento dell'architettura e del codice già presenti. Queste attività hanno riguardato la gestione del ciclo di vita delle chiavi, la rimozione della [DEK](#) globale diventata inutilizzata, la modifica al sistema di configurazione per permettere a database sullo stesso cluster di avere [KMS](#) differenti e la modifica al formato cifrato.

3.5.1 Rotazione delle chiavi

La rotazione delle chiavi crittografiche è un requisito fondamentale per l'adesione agli standard di sicurezza moderni.⁹

Nel sistema implementato è possibile a due livelli distinti della gerarchia delle chiavi crittografiche ovvero la rotazione della [DEK](#), che sostituisce la chiave di cifratura associata a una singola relazione, e la rotazione della [KEK](#), che aggiorna la chiave gestita dal [KMS](#) e richiede la ricifratura di tutte le [DEK](#) memorizzate nel catalogo.

⁹PCI Security Standards Council. *Payment Card Industry Data Security Standard (PCI DSS), Version 4.0*. https://www.pcisecuritystandards.org/document_library/. Requirement 3.5. 2022.

Rotazione della DEK

La rotazione della DEK implica la ricifratura dell'intero contenuto della tabella. Ogni riga memorizzata sul disco deve essere letta, decifrata con la chiave precedente, ricifrata con la nuova chiave e riscritta. Il modo più sicuro consiste nell'eseguire questa operazione all'interno di un'unica transazione con un blocco esclusivo sulla tabella. Questo garantisce che in caso di fallimento durante l'operazione l'intera relazione può essere portata allo stato precedente; tuttavia il blocco la rende inaccessibile per tutta la durata del processo, potenzialmente molto lunga su tabelle di grandi dimensioni.

Per superare la limitazione imposta dal blocco è stato progettato e implementato un meccanismo di rotazione *online*, `pg_vault_tde_rotate_online('table_name')`, la quale demanda l'operazione a un *background worker*¹⁰ lanciato dal processo *postmaster* in fase di avvio del server. La funzione, riservata al *superuser* del database, registra immediatamente una riga nella tabella di catalogo `pg_vault_tde_rotation_progress`, che riporta lo stato di avanzamento del processo di rotazione, e restituisce il controllo alla sessione chiamante senza attendere il completamento, permettendo così al database di rimanere pienamente accessibile durante la rotazione.

Il *background worker* opera in tre fasi, ciascuna racchiusa in una transazione. Nella prima, stima il numero totale di tuple leggendo il catalogo di sistema `pg_class` e aggiorna la tabella di avanzamento con lo stato `running`. Nella seconda fase, in cui avviene la rotazione effettiva, vengono sostanzialmente invocate sequenzialmente tre funzioni

- `pg_vault_tde_catalog_zero_rel_dek(relid)` la quale copia il valore dell'attributo `dek` in `prev_dek`, entrambi contenuti all'interno della *cache* in memoria condivisa, e marca l'attuale DEK non valida e la precedente come valida;
- `pg_vault_tde_catalog_update_rel_dek(relid)` che aggiorna la tupla all'interno della tabella di catalogo `pg_vault_tde_catalog` dell'estensione con una nuova DEK generata;
- `pg_vault_tde_reencrypt_table(relid)` che compie una ricifratura dell'intera tabella;

L'ultima funzione citata legge l'intera tabella tramite il metodo `table_beginscan` poiché invoca la *callback* `scan_getnextslot` sovrascritta da `encrypted_heap` che provvede a decifrare automaticamente la tupla. Successivamente la cifra con la DEK appena generata, disponibile in *cache* in quanto caricata al primo *miss*. Il blocco acquisito sulla tabella durante questa operazione è di tipo `RowExclusiveLock`, compatibile con tutti i blocchi non conflittuali,¹¹ come per esempio letture ed inserimenti.

Nella terza fase il record di avanzamento viene aggiornato allo stato `complete`.

¹⁰Documentazione PostgreSQL: Background Worker Processes. URL: <https://www.postgresql.org/docs/current/bgworker.html>.

¹¹Documentazione PostgreSQL: Explicit Locking. URL: <https://www.postgresql.org/docs/current/explicit-locking.html>.

Il meccanismo di decifratura durante la rotazione si basa sullo slot `prev_dek` nella struttura in *shmem*. Durante la finestra di rotazione, le righe cifrate con la vecchia chiave possono essere comunque decifrate correttamente, in quanto il **TAM** tenta prima la decifratura con la chiave corrente e, in caso di fallimento del tag di autenticazione GCM, prova con quella precedente.

L'avanzamento dell'operazione può essere monitorato in tempo reale dalla vista `pg_vault_tde_rotation_status`, che espone per ogni tabella lo stato corrente, il numero di tuple processate, il totale stimato e la percentuale di completamento.

Rotazione della KEK

La rotazione della **KEK**, gestita dalla funzione `pg_vault_tde_rotate_kek()`, non richiede la riscrittura dei dati nelle tabelle cifrate, ma soltanto la ricifratura di tutte le **DEK** presenti nel catalogo `pg_vault_tde_catalog` con la nuova **KEK**. Il comportamento concreto dipende dal provider **KMS** configurato, ma il processo è gestito in un solo punto nel codice grazie al fatto che il *provider* è rappresentato da una *vtable*. I puntatori ai metodi invocati durante questo processo sono:

- `*prepare_kek_rotation()` con cui il *provider* si prepara alla rotazione;
- `*commit_kek_rotation()` con cui il *provider* termina la rotazione;

Il processo di *rewrap*, ovvero di decifratura con la vecchia chiave e di cifratura con la nuova è operato dalla funzione `pg_vault_tde_catalog_rewrap_all(releid)` che a sua volta, per ogni riga della tabella di catalogo, invoca un altro metodo della *vtable*, `*rewrap_dek()` che compie l'operazione sopra descritta.

Nel caso del provider *vault*, la funzione di preparazione invoca l'endpoint di rotazione della chiave Transit di HashiCorp Vault o OpenBAO, in base al *endpoint url* configurato, il quale in fase di *rewrapping* è in grado di gestire la decifratura della *wrapped dek* con la vecchia chiave e la cifratura con la nuova, in modo indipendente.

Nel caso invece del provider locale, il *rewrapping* è maggiormente problematico, perché si rende necessario gestire la persistenza temporanea della **KEK** precedente che deve essere utilizzata per decifrare le **DEK** in catalogo. Questo viene reso possibile dall'introduzione di una *struct* che rappresenta il contesto di rotazione, definita come segue

```
typedef struct LocalKekRotationCtx
{
    unsigned char old_kek[TDE_DEK_LEN];
    unsigned char new_kek[TDE_DEK_LEN];
} LocalKekRotationCtx;

static LocalKekRotationCtx *local_kek_rotation_ctx;
```

Durante la preparazione alla rotazione viene generata una nuova **KEK** casuale, che non viene subito scritta nel file PKCS#12, in quanto un fallimento del processo risulterebbe

nella perdita della chiave attualmente memorizzata e quindi della possibilità di decifrare le tuple, ma memorizzata all'interno del contesto, insieme alla chiave da ruotare.

In questo modo la *routine* di *rewrap* è in grado di operare leggendo la struttura di contesto, la quale viene liberata e pulita mediante `OPENSSL_cleanse()` al *commit* della rotazione, momento in cui il file del wallet viene sovrascritto con la nuova chiave in quanto tutte le chiavi sono state ricifrate.

Al termine, indipendentemente dal [KMS](#) configurato, la cache condivisa in memoria viene svuotata per forzare il ricaricamento delle chiavi in formato *wrapped*, garantendo che tutti i *backend* operino con le versioni aggiornate.

3.5.2 Rimozione della DEK globale

L'architettura dell'estensione all'inizio del progetto prevedeva una [DEK](#) globale di *fallback* nel caso in cui non fosse stato possibile avere una chiave dedicata per ciascuna tabella. Tale chiave era condivisa tra tutte le tabelle cifrate all'interno dello stesso cluster PostgreSQL. La scelta di rimuoverla è stata dettata dal fatto che la compromissione di quest'ultima avrebbe determinato la perdita della sicurezza crittografica per tutte le relazioni ad essa associate. Di conseguenza, si è preferito interrompere il processo di creazione delle tabelle o di inserimento dei dati qualora non fosse possibile ottenere una [DEK](#) specifica per ciascuna relazione.

Questa transizione ha comportato la rimozione della struttura `TdeShmemData`, deputata a ospitare in memoria condivisa (*shared memory*) la [DEK](#) globale, identificata convenzionalmente dal valore `relid = 0`. Contestualmente, è stata eliminata dal catalogo `pg_vault_tde_catalog` la tupla preposta a tale chiave, precedentemente inserita in modo automatico durante la fase di avvio (*bootstrap*) del server.

Infine, l'adeguamento della suite di test per eliminare i riferimenti a questa chiave globale ha rappresentato l'attività più onerosa in termini di tempo e di *refactoring*.

3.5.3 Ottimizzazione delle performance

Analizzando le performance di un server PostgreSQL su cui era installata l'estensione `pg_vault_tde` si è notato che il tempo richiesto per completare uno *sequential scan* era 6 volte superiore rispetto a un'installazione PostgreSQL normale. Questo perché ogni tupla letta deve venire anche decifrata, ovvero deve essere conosciuta la sua [DEK](#) e deve avvenire l'effettiva decriptazione. Proprio queste due operazioni comportavano la causa maggiore della perdita di *performance*.

Conversione *TdeRelDekCache* in *hash map*

Per evitare di dover leggere ad ogni cifratura e decifratura la tabella di catalogo `pg_vault_tde_catalog` era già stata implementata una *cache* formata da una lista *TdeRelDekEntry*. Come citato in precedenza, l'accesso a tale lista non può essere randomico ma deve essere sequenziale, risultando in una complessità al più $O(n)$. Si è quindi scelto di convertire tale struttura dati in un'*hash map* definita come segue

```

typedef struct TdeRelDekMap
{
    Oid          relid;
    char         dek[TDE_DEK_LEN];
    char         prev_dek[TDE_DEK_LEN];
    uint64       generation;
    bool         dek_valid;
    bool         prev_dek_valid;
} TdeRelDekMap;

static HTAB      *rel_dek_map      = NULL;
static LWLock    *rel_dek_lock    = NULL;

```

la quale permette di ottenere una complessità di accesso media $O(1)$ mentre nel caso peggiore $O(n)$.

Una nota importante deve essere fatta sulle *map* nel linguaggio C, infatti queste non sono presenti nativamente in esso ma solitamente sono fornite da librerie esterne. Nel caso dell'estensione, l'implementazione che viene usata è quella fornita direttamente da PostgreSQL con HTAB.¹² Quest'ultima permette di creare una mappa a partire da una *struct* qualsiasi, interpretando il primo attributo di essa come la *hash key*.

Inoltre il lock non può più essere interno alla struttura ma deve essere comunque presente come puntatore statico a LWLock.

Infine, i metodi che prima accedevano alla lista in modo sequenziale sono stati riscritti in modo che utilizzino la funzione `hash_search()` fornita sia per le letture che per gli inserimenti, cambiando alcuni parametri flag che indicano l'operazione da eseguire.

Gestione del contesto crittografico

Quando una tupla deve essere cifrata o decifrata, le funzioni di OpenSSL che operano su di essa necessitano di un *contesto crittografico*, ovvero una struttura dati di tipo `EVP_CIPHER_CTX` il quale contiene informazioni come la chiave `DEK` o l'`IV` da utilizzare e può essere passato come parametro tra le varie funzioni.

Questa *struct* può essere creata con la funzione `EVP_CIPHER_CTX_new()`, inizializzata con chiave e `IV` mediante `EVP_EncryptInit_ex2()`, azzerata con `EVP_CIPHER_CTX_reset()` e liberata con `EVP_CIPHER_CTX_free()`.

In `pg_vault_tde` tale contesto era definito come

```

static EVP_CIPHER_CTX *enc_ctx = NULL;
static EVP_CIPHER_CTX *dec_ctx = NULL;

```

¹²Documentazione PostgreSQL: Struttura HTAB. URL: <https://doxygen.postgresql.org/structHTAB.html>.

Alla prima richiesta crittografica, il contesto corrispondente veniva creato, in modo che tutte le successive operazioni potessero richiamare solamente la funzione di *reset* o, in caso di errore, quella di deallocazione.

Un'analisi più approfondita delle funzioni di OpenSSL ha rivelato che invocare la funzione di *reset* in continuazione distrugge le performance, questo accade perché dopo l'azzeramento della *struct* la successiva chiamata di `EVP_EncryptInit_ex2()` deve ricalcolare il *key schedule*, ovvero le sotto chiavi tipiche dei cifrari a blocchi come AES, la quale è un'operazione computazionalmente costosa, e infine riallocare la memoria liberata dal *reset*. Proprio per questo la perdita di prestazioni si verificava in modo più visibile durante le operazioni crittografiche, come per esempio un *sequential scan*, compiute su tabelle con un numero di tuple maggiore. Per cui si è deciso di salvare il contesto con granularità di tabella e non più di singola operazione, nel senso che finché le operazioni crittografiche sono eseguite su tuple della stessa tabella con stessa generazione di chiave allora il contesto non viene in alcun modo azzerato, ma si provvede a cambiare solamente l'IV. A questo scopo è stata definita la seguente *struct*

```
typedef struct TdeCipherSlot
{
    EVP_CIPHER_CTX  *ctx;
    Oid              relid;
    uint64           generation;
} TdeCipherSlot;

static TdeCipherSlot tde_enc = { NULL, InvalidOid, 0 };
static TdeCipherSlot tde_dec = { NULL, InvalidOid, 0 };
```

Tale `EVP_CIPHER_CTX` viene resettato solamente quando l'id della relazione o il numero di generazione del chiamante sono diversi da quelli memorizzati.

Notiamo che il contesto `EVP_CIPHER_CTX` contiene al suo interno anche l'IV utilizzato per la cifratura o decifratura che, in AES-256-GCM, deve differire per ogni tupla.¹³ Si rende quindi necessario aggiornare tale attributo ad ogni operazione crittografica. Questa operazione è resa possibile dall'implementazione, della libreria OpenSSL, della funzione di inizializzazione `EVP_EncryptInit_ex2()` a cui è possibile fornire solamente il contesto e il nuovo IV per usare implicitamente l'ultima DEK fornita. Nel caso in cui quest'ultima sia assente, il metodo restituisce un errore.

```
EVP_EncryptInit_ex2(ctx, NULL, NULL, iv_ptr, NULL)
```

¹³National Institute of Standards and Technology. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D. Section 8. National Institute of Standards e Technology, 2007. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>.

3.5.4 Sistema di configurazione

In precedenza, la maggior parte dei parametri GUC utilizzava il contesto `PGC_POSTMASTER`¹⁴ il quale applica la configurazione in fase di avvio del server e non lo rende modificabile se non con un riavvio. I valori per questi parametri solitamente sono memorizzati nel file `postgresql.conf` o passati come parametro di linea di comando. Inoltre tale contesto fa riferimento a tutto il *cluster* PostgreSQL, per cui tutti i database accedono allo stesso valore senza possibilità di cambiarlo per se stessi. Questo rendeva impossibile ad esempio che due database nella stessa istanza del server potessero avere *KMS* distinti e di tipo diverso oppure per lo stesso database, di cambiare *url* del Vault a *runtime*. Per cui si è deciso di portare quasi tutti i parametri GUC a un livello di contesto con incidenza minore ovvero `PGC_SUSET` che permette al *superuser* di modificarli a livello di singolo database tramite `ALTER DATABASE SET`. Tale scelta consente a istanze PostgreSQL di configurare provider *KMS* differenti per database distinti senza necessità di riavvio del server. Il valore iniziale è comunque quello definito nel file di configurazione del *cluster*.

Fanno eccezione `pg_vault_tde.max_encrypted_relations` e `pg_vault_tde.crypto_provider`, che richiedono `PGC_POSTMASTER` poiché determinano rispettivamente la dimensione della struttura dati in memoria condivisa e il *provider* OpenSSL caricato all'avvio.

3.5.5 Modifica al formato cifrato

Durante lo sviluppo di `tde_btree` si è notato che gli indici su colonne di tuple cifrate non venivano aggiornati a seguito di un comando `UPDATE`. Il problema era legato al meccanismo degli *Heap-Only Tuples (HOT) updates*, progettato per saltare la reindicizzazione se le colonne indicizzate non subiscono modifiche. Per verificare l'assenza di modifiche, il sistema confronta la vecchia e la nuova versione della tupla. Tuttavia, nel caso delle righe cifrate, il confronto avveniva direttamente sui dati crittografati, poiché non era possibile decifrarli in quella fase. Inoltre, il recupero degli attributi da confrontare si basava su *offset* memorizzati nell'*header* della tupla, calcolati presumendo che i dati fossero in chiaro. Di conseguenza, quando la funzione di confronto cercava di accedere all'attributo tramite questi *offset*, finiva nella maggior parte dei casi per leggere i primi 9 byte del payload cifrato in quanto le colonne indicizzate solitamente si trovano in testa alla tupla, che però nel formato cifrato corrispondono ai campi costanti `VERSION_BYTE` e `GENERATION`. Essendo questi byte identici tra le due versioni della tupla, il sistema riteneva erroneamente che la colonna indicizzata fosse rimasta inalterata, saltando così l'aggiornamento dell'indice.

Per disabilitare il meccanismo di *HOT updates* si è reso necessario un *refactoring* del formato con il quale le tuple cifrate vengono memorizzate sul disco. Come descritto precedentemente, la cifratura introduce all'interno della tupla dei metadati necessari, come il

¹⁴Documentazione PostgreSQL: Vista `pg_settings`. URL: <https://www.postgresql.org/docs/current/view-pg-settings.html>.

Vettore di Inizializzazione (IV) e il *tag* di autenticazione. Il nuovo formato cifrato è stato quindi riordinato come

$$[\text{IV}(12) | \text{CT}(N) | \text{TAG}(16) | \text{VERSION_BYTE}(1) | \text{GENERATION}(8)]$$

in modo che l'*offset* di posizione di qualsiasi attributo non possa mai cadere nelle sezioni costanti del formato cifrato, infatti questo viene calcolato quando la tupla è ancora in chiaro quindi quando ha ancora lunghezza N, per cui gli *offset* in caso di tupla cifrata possono riferire solamente Byte nello spazio da 0 a N-12, i quali cambiano ad ogni cifratura.

In questo modo il meccanismo di HOT *updates* rileverà sempre una variazione di valori, attivando l'aggiornamento degli indici.

Ovviamente la disabilitazione di questa funzionalità di PostgreSQL comporta degli svantaggi come una maggiore pressione sul VACUUM il quale troverà molti più indici morti da eliminare, un *Index Bloat* generalmente più grande ed una pesantezza del database maggiore che rallenta le query. Tale soluzione infatti dovrebbe essere intesa come temporanea.

Capitolo 4

Verifica

La verifica della correttezza, ovvero l'accertamento che l'output di ogni modulo sia conforme e consistente rispetto alle specifiche decise in fase di progettazione, ha richiesto l'aggiornamento della suite di test di `pg_vault_tde`. L'architettura di questi si articola su più livelli: test di regressione SQL, test [TAP](#) in Perl, test di isolamento per la concorrenza e un benchmark di prestazioni.

4.1 Regressione

I test di regressione costituiscono la verifica funzionale principale. In totale sono presenti centoventotto test univoci. I primi settanta, in gran parte già presenti prima dell'inizio del tirocinio, coprono i percorsi fondamentali di cifratura e decifratura di tuple, assenza di testo in chiaro sul disco, aggiornamento con preservazione del *ctid* (*Current Tuple Identifier*), ovvero la posizione fisica della tupla, eliminazione, gestione di righe nulle, scansione tramite indice non criptato, inserimento massivo tramite `COPY FROM`, colonne di tipi misti, rotazione della chiave con verifica del rifiuto delle versioni cifrate precedenti, e tutti i percorsi di accesso al dato: `BitmapHeapScan`, `TABLESAMPLE`, `SELECT FOR UPDATE`, `ANALYZE`.

A questi sono stati aggiunti i test specifici per le funzionalità sviluppate durante lo stage come la riscrittura completa della tabella tramite `VACUUM FULL` e `CLUSTER`, cifratura e decifratura durante operazioni [Data Query Language \(DQL\)](#) con attributi [TOAST](#), la verifica del corretto *access method* della *toast table*, la cifratura deterministica degli indici B-Tree (`tde_btree`) su colonne di tipo a lunghezza fissa e variabile con i rispettivi operatori di uguaglianza, la validazione della modifica a *runtime* dei parametri di configurazione GUC, la rotazione *online* della [DEK](#) tramite *background worker*, la rotazione della [KEK](#) con conseguente *re-wrapping* delle chiavi nel catalogo e l'invalidazione della *cache* delle chiavi.

A titolo esemplificativo viene riportato il test di verifica del funzionamento dell'operazione di `VACUUM FULL` su tabelle con *storage* di tipo *external*.

```
-- =====
-- TEST 109: VACUUM FULL ON TABLE WITH EXTERNAL STORAGE COLUMNS
```

```
-- =====

DO $$
DECLARE
    rel_id oid;

BEGIN
    DROP TABLE IF EXISTS test_table_109;
    CREATE TABLE test_table_109 (
        id serial,
        value text
    ) USING encrypted_heap;

    ALTER TABLE test_table_109 ALTER COLUMN value SET STORAGE EXTERNAL;

    INSERT INTO test_table_109 (value) VALUES (repeat('ciao_ciao', 10000));
END;
$$;

SELECT pg_stat_force_next_flush();
VACUUM FULL test_table_109;

DO $$
DECLARE
    v_len    int;
    v_val    text;
    expected text := repeat('ciao_ciao', 10000);
BEGIN
    SELECT length(value) INTO v_len FROM test_table_109 WHERE id = 1;
    IF v_len IS DISTINCT FROM length(expected) THEN
        RAISE EXCEPTION 'TEST 109 FAILED: length mismatch after
                        VACUUM FULL (got %, expected %)',
                        v_len, length(expected);
    END IF;

    SELECT value INTO v_val FROM test_table_109 WHERE id = 1;
    IF v_val IS DISTINCT FROM expected THEN
        RAISE EXCEPTION 'TEST 109 FAILED: content mismatch after VACUUM FULL';
    END IF;

    DROP TABLE IF EXISTS test_table_109;
```

```

RAISE NOTICE 'TEST 109 PASSED: VACUUM FULL on STORAGE EXTERNAL
              column decrypts correctly after rewrite';
END;
$$;

```

4.2 Isolamento

I test di isolamento, scritti nella sintassi dello *isolation tester*¹ di PostgreSQL, verificano la correttezza del comportamento in presenza di transazioni concorrenti in particolare che la rotazione *online* della chiave non blocchi le letture o le scritture in corso e che la visibilità delle tuple cifrate sia preservata in sessioni multiple.

La potenza dello strumento fornito da PostgreSQL è quella di permutare le singole operazioni delle transazioni concorrenti per verificare tutti i casi di isolamento possibili. Ad esempio:

```

permutation "su_setup" "rx_begin" "rx_read1" "rot_rotate"
              "wx_begin" "wx_write" "wx_commit" "rx_read2" "rx_commit"
permutation "su_setup" "wx_begin" "wx_write" "rot_rotate" "wx_commit"
              "rx_read2"
permutation "su_setup" "wx_begin" "wx_write" "rot_rotate" "rx_begin"
              "rx_read2" "wx_commit" "rx_read2" "rx_commit"
permutation "su_setup" "wx_begin" "wx_update" "rot_rotate" "wx_abort"
permutation "su_setup" "vacuum" "rot_rotate"
permutation "su_setup" "rot_rotate" "vacuum"

```

dove ogni termine scritto tra virgolette identifica un'operazione definita dal programmatore in fase di *setup* del test.

4.3 TAP

I test **TAP** sono scritti in linguaggio Perl tramite il framework `PostgreSQL::Test::Cluster`. Di questi sono presenti undici test suite per un totale di centoundici test singoli.

I primi test verificano il caricamento base dell'estensione.

La restante parte si è resa fondamentale per la verifica del sistema di backup logico `pg_dump_tde`. La suite valida l'esportazione, il formato del file come visibile in [Figura 3.3](#) e il ripristino con entrambi i provider KMS. Vengono testati sia il *wallet* locale sia l'integrazione con HashiCorp Vault tramite un *container* Podman. Si verificano inoltre esportazioni su database che differiscono da quello di backup.

¹ *PostgreSQL Isolation Tester README*. URL: <https://github.com/postgres/postgres/blob/master/src/test/isolation/README>.

Il formato crittografico di AES-256-GCM e il *tag* di autenticazione sul numero di sequenza del blocco vengono testati estensivamente introducendo alterazioni sul file cifrato, modifiche di posizione dei blocchi e troncamenti. Il ripristino in queste condizioni produce un errore deterministico di autenticazione del tag GCM.

Sono validati anche i tentativi di ripristino con chiavi crittografiche errate e le opzioni della *command line interface*.

Infine, l'ultimo file verifica il corretto funzionamento di architetture con provider KMS diversi all'interno dello stesso *cluster*.

4.4 Performance

Al fine di misurare l'*overhead* che la [TDE](#) aggiunge alla normale esecuzione di PostgreSQL sono stati implementati alcuni test di *benchmark*, i quali misurano, in percentuale, quanto maggiore è il tempo richiesto per portare a termine inserimenti, scansioni sequenziali, scansioni con indici ed aggiornamenti massivi.

La misurazione del tempo richiesto dalle operazioni è resa possibile dal comando PostgreSQL `EXPLAIN ANALYZE`.² Invece per le `INSERT` viene fatto con il comando `bash date +%s%N`.

Sono presenti tre profili di utilizzo del database, ognuno dei quali viene eseguito con tabelle di un milione di righe per tre iterazioni. È possibile modificare tali parametri di esecuzione ad esempio aumentando il numero di iterazioni per ottenere dei risultati più precisi richiedendo un tempo maggiore.

I profili sono i seguenti:

- *tiny*, con righe di dimensione media di 32 byte;
- *oltp*, con righe di dimensione media di 256 byte;
- *wide*, con righe di dimensione media di 512 byte.

dove *oltp* si riferisce a database di tipo [Online Transaction Processing \(OLTP\)](#).

L'*overhead* medio misurato su operazioni con dieci milioni di righe e dieci iterazioni è stato circa del 104%, quindi poco più del doppio rispetto a una normale installazione di PostgreSQL.

²Documentazione PostgreSQL: Using `EXPLAIN`. URL: <https://www.postgresql.org/docs/current/using-explain.html>.

Capitolo 5

Conclusioni

5.1 Limitazioni

Tutti gli obiettivi stabiliti nel Piano di Lavoro del tirocinio sono stati raggiunti con l'eccezione dei test di *Disaster Recovery*, che non è stato possibile implementare in quanto è stata data maggiore importanza ad altre funzionalità e alla loro conclusione.

Nonostante questo, l'implementazione attuale di `pg_vault_tde` presenta alcune limitazioni architetturali e funzionali, in parte dovute ai vincoli del sistema di estensioni di PostgreSQL:

- **Index-Only Scan e Range Scan:** Non sono supportati gli *Index-Only Scan* per gli indici `tde_btree`. Inoltre, le *Range Scan* sugli indici non sono possibili in quanto restituiscono risultati vuoti poiché l'algoritmo AES-SIV non preserva l'ordine. Rimangono comunque possibili attraverso *sequential scan*;
- **Cifratura del WAL:** L'estensione non è in grado di cifrare interamente i [Write-Ahead Log \(WAL\)](#), ma i dati riportati in essi appaiono come *ciphertext*;
- **Cursori WITH HOLD:** I dati temporanei generati da un cursore `WITH HOLD`¹ che eccedono la `work_mem` vengono scritti su disco in chiaro. L'executor scrive questi dati bypassando la *callback* del [TAM](#), impedendo all'estensione di cifrarli. Il bug è stato scoperto in fase di sviluppo ma, insieme al tutor aziendale, è stato deciso per il momento di assegnargli minore priorità in quanto rappresenta un rischio latente;
- **Heap-Only Tuples Updates:** Gli aggiornamenti [HOT](#) sono disabilitati *by design*. Poiché il formato del testo cifrato inizia con il [IV](#) casuale, ogni aggiornamento di qualsiasi attributo della tupla viene rilevato come un aggiornamento totale, impedendo a PostgreSQL di seguire il percorso ottimizzato [HOT](#) ma garantendo la coerenza degli indici.

¹Documentazione PostgreSQL: *define a cursor*. URL: <https://www.postgresql.org/docs/current/sql-declare.html>.

5.2 Valutazione personale

Personalmente mi ritengo molto soddisfatto dell'esperienza di tirocinio svolta presso Miriade Srl, in quanto mi ha permesso di approfondire a un livello di dettaglio molto elevato PostgreSQL e il suo funzionamento. Inoltre, ho avuto l'occasione di apprendere alcune delle *best practice* della programmazione in linguaggio C per sistemi critici in cui la disponibilità, reattività e integrità sono fondamentali. Ho avuto modo di approfondire gli strumenti a supporto dello sviluppo per un progetto esteso come i *build tool* e la costruzione di infrastruttura di test con *Podman*. Ho avuto modo di sperimentare i metodi di lavoro agile, finora studiati solamente dal punto di vista teorico, ed il lavoro in *team* attraverso le cerimonie di *Scrum*.

Inoltre, durante lo svolgimento dello stage ho avuto l'opportunità di mettere in pratica e allenare le mie capacità di organizzazione del lavoro e comunicazione con gli *stakeholder*, ruolo ricoperto in primis dal tutor aziendale. Infatti, durante le riunioni previste dal metodo di lavoro dovevo comunicare, nel modo più chiaro e conciso possibile, i risultati raggiunti, gli obiettivi fissati e le difficoltà riscontrate.

Ritengo di dover ringraziare tutte le persone di Miriade che mi hanno seguito durante questo percorso, in particolare Matteo Durighetto, tutor aziendale dello stage e *Head of R&D*, e Giulia Erica Valente, *Head of Data Architecture* e responsabile dell'area *Data*.

Acronimi e abbreviazioni

- AAD** [Additional Authenticated Data](#). 25, 48
- API** [Application Program Interface](#). 2, 3, 5, 15, 48
- DBMS** [Database Management System](#). 2–5
- DEK** [Data Encryption Key](#). 9, 10, 13–15, 25, 29, 32–37, 41, 48
- DQL** [Data Query Language](#). 41, 48
- GDPR** [General Data Protection Regulation](#). 2
- HOT** [Heap-Only Tuples](#). 38, 39, 45
- IAM** [Index Access Method](#). 9, 12, 26, 48
- IPC** [Inter-Process Communication](#). 26, 48
- ITS** [Issue Tracking System](#). 6, 48
- IV** [Initialization Vector](#). 11, 26, 29, 36, 37, 39, 45, 49
- KEK** [Key Encryption Key](#). 9, 10, 13, 15, 25, 32, 34, 41, 49
- KMS** [Key Management System](#). 2, 10, 13, 15, 25, 32, 34, 35, 38, 49
- OLTP** [Online Transaction Processing](#). 44, 49
- TAM** [Table Access Method](#). 9–12, 19, 20, 23, 26, 29, 30, 34, 45, 49
- TAP** [Test Anything Protocol](#). 5, 26, 41, 43, 49
- TDE** [Transparent Data Encryption](#). 2, 3, 23, 26, 44, 49
- TOAST** [The Oversized-Attribute Storage Technique](#). 9, 12, 19, 20, 41
- WAL** [Write-Ahead Log](#). 45, 49

Glossario

AAD dati aggiuntivi forniti ad un algoritmo crittografico a cifratura autenticata (come AES-GCM) che vengono autenticati per garantirne l'integrità, ma non cifrati. Vengono tipicamente impiegati per associare metadati (come numeri di sequenza o identificativi) al payload cifrato, prevenendo attacchi di manipolazione, riordinamento o troncamento. [47](#)

API in informatica, con il termine *Application Programming Interface* si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra il software a basso livello e quello ad alto livello, semplificando così il lavoro di programmazione. [47](#)

DEK chiave di cifratura utilizzata per cifrare direttamente i dati, nel contesto specifico le tuple della base di dati. [47](#), [49](#)

DQL sottoinsieme del linguaggio SQL utilizzato per interrogare e recuperare i dati da una base di dati. L'istruzione principale è `SELECT`. [47](#)

IAM interfaccia che definisce il metodo con cui il database interagisce con i dati attraverso l'utilizzo di un indice. Permette di creare un nuovo tipo di indice, definendone il comportamento alla sua creazione, all'inserimento dei dati nella tabella associata o alla loro eliminazione. [47](#)

IPC in informatica si riferisce all'insieme di tecnologie, meccanismi e protocolli che consentono a processi o thread diversi, in esecuzione sullo stesso sistema operativo o su sistemi diversi, di comunicare. Alcuni esempi sono le pipe, i socket e la memoria condivisa. [47](#)

ITS applicazione software che consente di registrare, gestire e monitorare lo stato di avanzamento dello sviluppo di singole funzionalità e dei bug. Fornisce un ambiente centralizzato per la collaborazione del team di sviluppo e per la tracciabilità delle modifiche. [47](#)

- IV** blocco di bit utilizzato dagli algoritmi crittografici per garantire che testi in chiaro identici, cifrati con la stessa chiave, producano *ciphertext* differenti. 47
- KEK** chiave di cifratura utilizzata per cifrare altre chiavi crittografiche, nello specifico le DEK. 47, 49
- KMS** sistema per la gestione centralizzata del ciclo di vita delle chiavi crittografiche, utilizzato nel contesto specifico per la gestione delle chiavi KEK. 47
- OLTP** categoria di sistemi software progettata per l'elaborazione di transazioni in tempo reale. Sono ottimizzati per gestire in parallelo un elevato numero di operazioni brevi come lettura, inserimento, aggiornamento, cancellazione, garantendo bassa latenza e forte consistenza nel rispetto delle proprietà ACID. 47
- TAM** interfaccia che definisce il metodo con cui il database interagisce direttamente con i dati memorizzati sul disco. Permette di modificare il comportamento del database all'inserimento, lettura e modifica dei dati. 47
- TAP** interfaccia testuale utilizzata per la comunicazione tra test e un sistema di orchestrazione (test harness).² Nel contesto di PostgreSQL è impiegato per gestire scenari di collaudo complessi che richiedono il controllo del ciclo di vita di intere istanze del database. 47
- TDE** tecnologia crittografica impiegata per cifrare i dati a riposo (data-at-rest) all'interno di un database in modo del tutto trasparente per le applicazioni. 47
- WAL** meccanismo standard per garantire l'integrità dei dati nei sistemi di gestione dei database transazionali. Tutte le modifiche ai dati vengono registrate in log sequenziali prima di essere scritte nei file di dati, garantendo la *crash recovery* e l'atomicità delle transazioni. 47

²Test Anything Protocol. URL: <https://testanything.org/>.

Bibliografia

Riferimenti bibliografici

Silberschatz, A., H. F. Korth e S. Sudarshan. *Database System Concepts*. 7^a ed. New York, NY, USA: McGraw-Hill, 2019.

Siti web consultati

Chapter 36: Extending SQL. URL: <https://www.postgresql.org/docs/18/extend.html> (cit. a p. 3).

Documentazione HashiCorp Vault. URL: <https://developer.hashicorp.com/vault/docs> (cit. a p. 16).

Documentazione PostgreSQL. URL: <https://www.postgresql.org/docs/current/xfunc-c.html> (cit. a p. 4).

Documentazione PostgreSQL pg_dump. URL: <https://www.postgresql.org/docs/current/app-pgdump.html> (cit. a p. 23).

Documentazione PostgreSQL TAP Tests. URL: <https://www.postgresql.org/docs/current/regress-tap.html> (cit. a p. 5).

Documentazione PostgreSQL TOAST. URL: <https://www.postgresql.org/docs/current/storage-toast.html> (cit. a p. 20).

Documentazione PostgreSQL VACUUM. URL: <https://www.postgresql.org/docs/current/sql-vacuum.html> (cit. a p. 22).

Documentazione PostgreSQL: Background Worker Processes. URL: <https://www.postgresql.org/docs/current/bgworker.html> (cit. a p. 33).

Documentazione PostgreSQL: define a cursor. URL: <https://www.postgresql.org/docs/current/sql-declare.html> (cit. a p. 45).

Documentazione PostgreSQL: Explicit Locking. URL: <https://www.postgresql.org/docs/current/explicit-locking.html> (cit. a p. 33).

- Documentazione PostgreSQL: Grand Unified Configuration (GUC)*. URL: <https://www.postgresql.org/docs/current/config-setting.html> (cit. a p. 10).
- Documentazione PostgreSQL: Hash Indexes*. URL: <https://www.postgresql.org/docs/current/indexes-types.html#INDEXES-TYPES-HASH> (cit. a p. 28).
- Documentazione PostgreSQL: Interfacing Extensions to Indexes*. URL: <https://www.postgresql.org/docs/current/xindex.html> (cit. a p. 27).
- Documentazione PostgreSQL: LWLocks*. URL: <https://www.postgresql.org/docs/18/xfunc-c.html#XFUNC-ADDIN-LWLOCKS> (cit. a p. 14).
- Documentazione PostgreSQL: Shared Memory*. URL: <https://www.postgresql.org/docs/current/xfunc-c.html#XFUNC-SHARED-ADDIN> (cit. a p. 14).
- Documentazione PostgreSQL: Struttura HTAB*. URL: <https://doxygen.postgresql.org/structHTAB.html> (cit. a p. 36).
- Documentazione PostgreSQL: Using EXPLAIN*. URL: <https://www.postgresql.org/docs/current/using-explain.html> (cit. a p. 44).
- Documentazione PostgreSQL: Vista pg_settings*. URL: <https://www.postgresql.org/docs/current/view-pg-settings.html> (cit. a p. 38).
- Eisentraut, Peter. *Transparent column encryption*. 2021. URL: <https://www.postgresql.org/message-id/flat/89157929-c2b6-817b-6025-8e4b2d89d88f@enterprisedb.com> (cit. a p. 2).
- Gitflow*. URL: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> (cit. a p. 6).
- Harkins, Dan. *RFC 5297: Synthetic Initialization Vector (SIV) Block Cipher Mode for Key Wrapping and Authenticated Encryption*. 2008. URL: <https://datatracker.ietf.org/doc/html/rfc5297> (cit. a p. 26).
- Miriade Srl - Sito Ufficiale*. URL: <https://www.miriade.it> (cit. a p. 1).
- Moon, Insung. *[Proposal] Table-level Transparent Data Encryption (TDE) and Key Management Service (KMS)*. 2018. URL: [https://www.postgresql.org/message-id/flat/031401d3f41d\\$5c70ed90\\$1552c8b0\\$@lab.ntt.co.jp](https://www.postgresql.org/message-id/flat/031401d3f41d$5c70ed90$1552c8b0$@lab.ntt.co.jp) (cit. a p. 2).
- PostgreSQL Isolation Tester README*. URL: <https://github.com/postgres/postgres/blob/master/src/test/isolation/README> (cit. a p. 43).
- Repository pg_tde di Percona*. URL: https://github.com/percona/pg_tde (cit. a p. 3).
- Sito PostgreSQL*. URL: <https://www.postgresql.org/> (cit. a p. 5).
- Test Anything Protocol*. URL: <https://testanything.org/> (cit. a p. 49).